UNIVERSITY OF OSLO
Department of Informatics

Implementing Rate Control in NetEm

Untying the NetEm/tc tangle

Master thesis

Anders G. Moe

19th August 2013

Implementing Rate Control in NetEm

Anders G. Moe

19th August 2013

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisors, Andreas Petlund, Pål Halvorsen, Carsten Griwodz, for a lot of help with my Thesis, and providing valuable feedback. Thanks to all the guys at the lab, for good times and great help. Thanks to Jonas for providing feedback and numerous trips to "the Dane".

Abstract

Simulation and emulation are techniques often used in the research community. They can be used during development, testing and debugging of a new protocol, to test and look at the characteristics of a protocol already in use. And to evaluate the overall performance of the network.

Network emulation is used to test real network systems where we can configure and control the environment. Network emulation lets us process real network traffic from the emulated network by using traffic shapers to vary common network parameters in a controlled way. Some of the network behaviour that we can control this way are delay, loss, jitter and bandwidth. This helps when we want to test different protocols or applications under different circumstances.

An emulator can also be hard to configure in the correct manner, and a faulty configuration can lead to incorrect results. NetEm, a network emulator included in the Linux kernel doesn't have a built-in bandwidth emulator, but relies on other queueing disciplines in the kernel to do bandwidth limitation. Configuring these together is very hard to get right. In this thesis we propose a design for a built-in rate emulator extension to NetEm, which we also implement. This implementation is then tested to verify the condition of our design.

Chapter 1

Introduction

1.1 Background

Simulation and emulation are techniques often used in the research community. They can be used during development, testing and debugging of a new protocol, to test and look at the characteristics of a protocol already in use, and evaluate the overall performance of the network.

A network simulator is a program that creates abstractions of reality. This is used to mimic the operations of a real network, from the physical link (cables or wifi) between two or more nodes, through layers, all the way up to an application. This means that everything will be done in software, and no real network traffic is generated.

Network emulation is used to test real network systems where we can configure and control the environment. In comparison with network simulators, where no real network traffic is generated, network emulation lets us process real network traffic from the emulated network. In network emulation we use traffic shapers to vary usual network parameters in a controlled way. Some of the network behaviour that we can control in this way are delay, loss, jitter and bandwidth. This helps when we want to test different protocols or applications under different circumstances. A small drawback with emulation is that we can usually just test end-to-end behaviour as the emulator acts as the network cloud between the end nodes.

1.2 Problem Definition

Netem is an emulation tool for linux where we can emulate network traffic like delay, loss and jitter. It is used in conjunction with traffic control (TC) that can be used to add bandwidth limitation. When using these together on the same network interface we have observed experiments that give strange results. We found the configuration of rate control in conjunction with NetEm to be the problem, and that the general configuration of two

queueing disciplines is very hard to get right. We have therefore decided to focus this thesis on proposing a design and implementing a built-in rate control extension to NetEm. With a working implementation, we will run extensive tests to verify our implementation.

1.3 Contributions

There is no doubt that the TC/NetEm combination is very hard to configure correctly. With NetEm being used heavily in the research community, this could lead to faulty research. As part of this thesis we have designed and implemented a built-in rate control extension to NetEm. This design aims to remove the three problems we have identified when using TC and NetEm: Configuration of the system is hard, it uses one queue that must handle both the packets that are delayed as a cause of rate emulation and the packets delayed because of latency emulation. The third problem is the fact that the rate controls implemented in the other queueing disciplines have not been developed for network emulation. They have all been implemented as parts of a traffic control scenario. We have tested the implementation and have concluded that it gives good results. It is easier to configure as it is only one queueing discipline, it makes use of two queues to split the rate control from the delay emulation, and the rate control is focused on eliminating bursts higher than the configured rate.

1.4 Outline

This document is organized as follows. Chapter two is dedicated to background information that is relevant to this thesis. Some general information about system evaluation, a deep look at Linux networking with a focus on traffic control and an explanation of the NetEm and Dummynet network emulators. In chapter three we first look at the problem of TC and NetEm before we propose a design of a new version of NetEm. This chapter also includes the implementation of said design. In chapter 4 we present the results from testing the new implementation, and we conclude the thesis in chapter 5.

Chapter 2

Background

This chapter presents background information necessary to understand this thesis. It will briefly explain how systems can be evaluated, with a focus on network emulation. Linux kernel networking will be explained, followed by an in in-depth view of traffic control, with a heavy focus on Linux queueing disciplines. In the end the network emulators, NetEm and Dummynet will be explained.

2.1 System evaluation

As we develop new applications and protocols for the Internet, we need a reliable and realistic way to analyse and test them with regard to their impact on the rest of the network and their performance. The techniques used are usually divided up in three types. Analytical modelling, simulation and measurements[43]. Analytical techniques are theoretical and can therefore be used in the early stages of development. Usually a lot of simplifications and assumptions are needed and the results will often be varying in comparison to reality. Simulation techniques will make use of programs that simulate the real world. This will therefore be closer to reality than analytical evaluation, but it will still not be as accurate as testing the real system as we have to abstract from the details so that it will be cost efficient to implement. The last technique, measurement, is the technique where we investigate the real system. To use this technique the system must obviously already be implemented, and that means that it cant be used until we have a working application or protocol.

Theoretical analysis makes it possible to explore a model of a target network with complete control, which can give a basic understanding of how a new protocol or application will work. It provides a basis to check the design, and can help with identifying problems at an early stage. As a lot of simplifications and assumptions are needed, at least with more complex systems, we have a huge risk of using a model that is so simple that the most important behaviours of the real system is gone. The results

of the theoretical work can then be at worst useless, and that can be dangerous as it might not be so easy to notice.

Simulation can be used to check new protocols or applications behaviour and characteristics in varying network conditions, and possible impacts they can have on the target network. Simulations are done by implementing a model of the target network in the simulators software, some models are the same as in the analytical stages, but most are usually more complex. With the use of more complex models in the simulator, it will be possible to check if the simplifications done in the analytical step have invalidated the model. If so, the results of the analysis might not be correct with the intended model. Simulators is not only a way to check the validity of an analysis, but can also be used to explore other complex scenarios that might be impossible to do with theoretical analysis.

While we use analysis and simulation in the earlier stages of development, we have to complement the theoretical results with measurements once we have an implementation ready. The measurements can be gathered by testing the application or protocol in a target network. If the target network is large, for example the Internet, it will be costly and difficult to implement a controllable and repeatable environment to do the testing. One possible solution is to use network emulation, which makes it possible to use real hosts with real protocols and applications to do the testing in a controlled environment.

2.2 Linux Networking

To explain the networking part of the Linux kernel, we will go through a short explanation of the sk_buff data structure. It minimizes copying overhead as packets flow through the different networking layers as it is used as a common data structure to manage network data and headers. Following this, we will explain the packet flow in the Linux kernel through the transmission and receive paths. To explain the network flow we will use the 2.6.20 version of the kernel, which is quite old, as this have the best documentation available that we have found [10]. Some function calls and names have changed in later versions of the kernel, but the flow remains the same, so this has no impact for the general understanding needed for this thesis.

2.2.1 sk_buff

An important part of the Linux kernel when it comes to networking is the common data structure *struct sk_buff* [12]. This struct is used in all network queues and buffers and is large enough to contain all the control information needed for a packet. Instances of *sk_buff* is almost exclusively called referred to as *skb*. Some of the important fields included in a skb is:

- *next:* Pointer to the next sk_buff in a list
- *prev:* Pointer to the previous sk_buff in a list
- *dev:* The device the packet arrived on or are leaving by
- *sk:* The socket the packet is owned by
- *transport_header:* The transport layer header
- *network_header:*The network layer header
- *mac_header:* The link layer header
- *data:* Pointer to data head

The first two fields in a sk_buff is next and prev. These are sk_buff pointers and they are used to organize skbs in a doubly linked list. The *sk_buff_head* struct is used to represent queues. This struct also contains *next* and *prev* sk_buff pointers, which is used to point to the first and the last sk_buffs in its queue. This makes it very efficient to add and remove skbs from a queue and moving them between queues. This single representation of a packet is also a very efficient way for moving packets between the different networking parts of the kernel.

2.2.2 Packet flow in Linux

Packets being sent or received have to go through the Linux networking kernel, and this section will explain packet flows through the networking kernel through some simplified examples. Packets being sent go through the transmission path while packets being received will go through the receive flow. A connection using Transmission Control Protocol (TCP)[30] over Internet protocol v4 (IPv4)[29] over Ethernet will be used as the example as this is the most common form of network connections. There are, of course, many other combinations possible using the different layer protocols. The kernel makes heavy use of virtual methods through function pointers as will be seen below. As a result of this there are a huge amount of possibilities when a packet is flowing through the kernel. To simplify the example, some function calls and alternative paths, that have no impact on this thesis, have been removed from this example.

Transmission path

The transmission path will be explained by following the path the data takes from the application layer all the way down to the link layer where it will be sent out on the physical layer. An overview of the transmission path is shown in figure 2.1

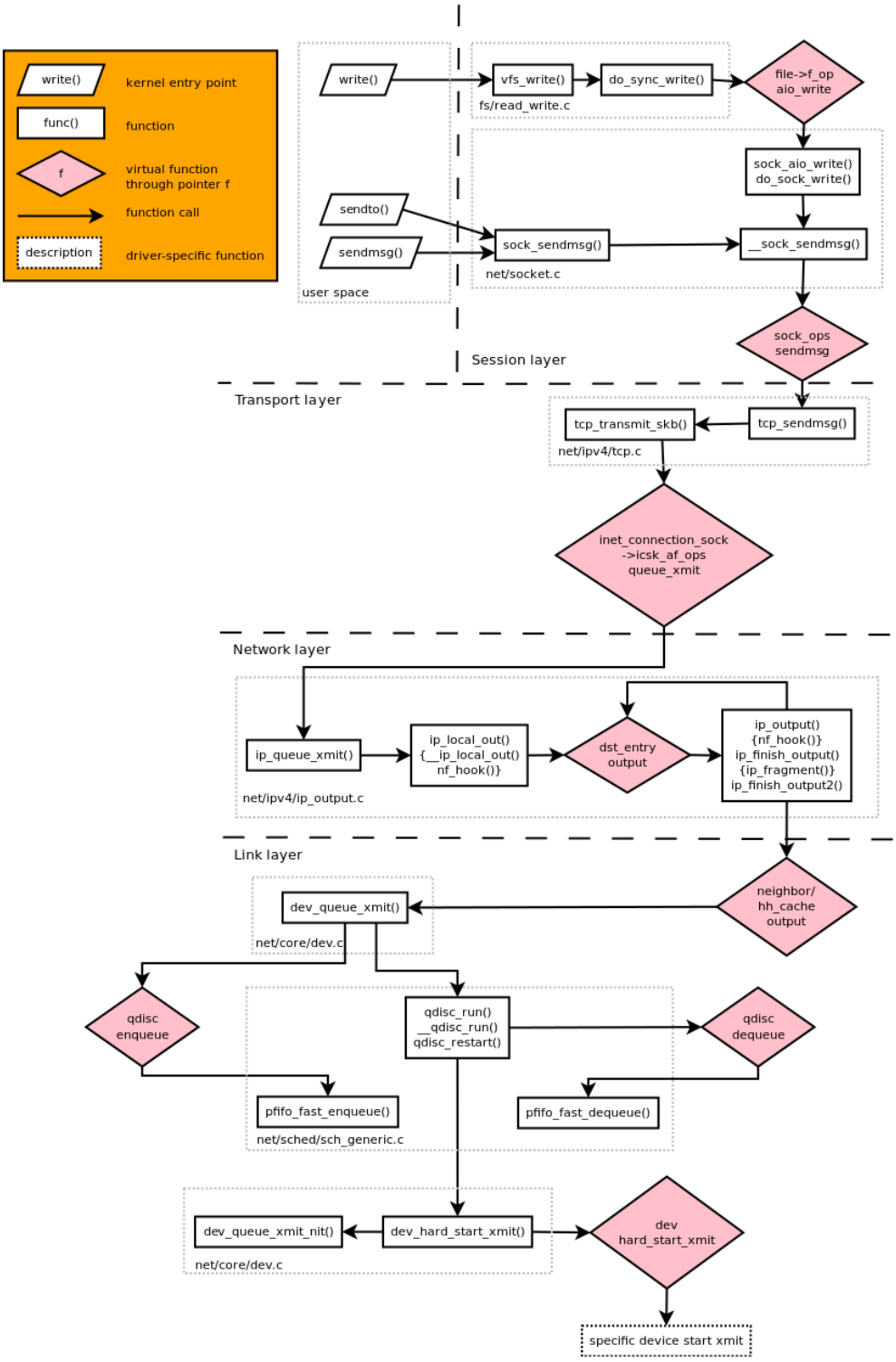


Figure 2.1: Transmission path

- **Layer 5: Session layer (sockets and files)**

  The application layer have three system calls it can use to send data over the network: *write()*, *sendto()* and *sendmsg()*. All three will eventually end up in *__sock_sendmsg()*. This method will check permissions with *security_sock_sendmsg()*, before it forwards the message to the next layer by calling the socket's sendmsg virtual method.

- **Layer 4: Transport layer (TCP)**

  In this example, the socket's virtual sendmsg method is *tcp_sendmsg()*. For each segment in the message, *tcp_sendmsg()* will first find an sk_buff which has space available or allocate a new one. Then it will copy the data from user space to the sk_buff data space using *skb_add_data()*. At this point there is a possibility for segmentation if the write is too large, or coalescing of individual writes if they are small enough to fit in one sk_buff. The data that ends up in the same sk_buff will become a single TCP segment. Still, the segments might be further fragmented at the network layer. After this the TCP queue is activated, and for each packet a call will be made to *tcp_transmit_skb()*. This method builds the TCP header in the space left by the allocation of the sk_buff, and clones the skb so it can pass control to the network layer. The call to the network layer is through the queue_xmit virtual method of the socket's address family, which in this case is AF_INET for IPv4 and its method *ip_queue_xmit()*.

- **Layer 3: Network layer (IPv4)**

  ip_queue_xmit() will do routing if necessary and create the IPv4 header. The routing decision made here will create a destination object called dst_entry. To perform actual output a call will be made to the dst_entry's output virtual method. In this example the sk_buff will be sent to *ip_output()* which will do post-routing filtering, in the case of netfiltering it re-outputs it on a new destination, fragments the datagram into packets if necessary, and in the end sends it to the output device. In the case of TCP the fragmentation step should not be needed as TCP already makes sure that packets are smaller than the maximum transmission unit (MTU). The output is given to the link layer with another virtual method call, which is usually *dev_queue_xmit()*. It is worth mentioning that *nf_hook()* will be called in several places at the network layer. This hook may modify or discard the datagram to perform network filtering like firewall, nat, etc.

- **Layer 2: Link layer (Ethernet)**

  At the link layer the kernel's main function is to schedule packets to be sent out on the physical layer. Linux uses the queueing discipline

(qdisc) abstraction to do this. Queueing disciplines are explained in detail in section 2.3.2. When a packet arrives at the link layer the *dev_queue_xmit()* method will put the sk_buff on the device queue using the qdisc->enqueue virtual method. In this case the default linux qdisc is used so the method called will be *pfifo_fast_enqueue()* Then the device output queue will be called immediately with *qdisc_run()*. This method will call *qdisc_restart()*, which uses the qdisc->dequeue virtual method to take an skb from the queue to be sent out on the network. The skb is then sent with *dev_hard_start_xmit()* and removed from the qdisc. If the sending fails, the skb will be re-queued in the qdisc and *netif_schedule()* is called to schedule a retry. This raises a software interrupt that calls *net_tx_action()*, which calls *qdisc_run()* for all devices with an active queue. When a packet is to be sent with *dev_hard_start_xmit()* it will first use *dev_queue_xmit_nit()*, which will check if the packet handler has been registered for the ETH_P_ALL protocol that is used for tcpdump. Then it will call the device driver's hard_start_xmit virtual function, which in turn will generate one or more commands to the network device to transfer the buffer. After some time it replies that it is done and will trigger the freeing of the sk_buff.

Receive flow

The receive flow will be explained by following data from when it enters the link layer all the way up to the session layer where its delivered to the application layer. An overview of the receive flow is shown in figure 2.2

- **Layer 2: Link layer (Ethernet)**

  The network device will have pre-allocated buffer space for reception, and the device's interrupt handler will just call *netif_rx_schedule()* and return from the interrupt. *netif_rx_schedule()* will add the device to softnet_data's poll_list and raise a software interrupt. The poll virtual method of the device will then be called by *net_rx_action()* which is run by *ksoftirqd* in the interrupt. This poll method will do device specific buffer management and will call *netif_receive_skb()* for all sk_buffs waiting to be received by the link layer. *netif_receive_skb()* is the method that will deliver the sk_buff the packet handler where it is supposed to go next. For example, it could be to the ETH_P_ALL protocol used by tcpdump, to the handlers for ingress queueing, handling for bridging or the packet handler registered for the network layer protocol specified by the packet. In this example it will be to the packet handler registered by IPv4. All packet handlers will be called through the *deliver_skb()* function, which in turn calls the virtual method of the desired protocol.

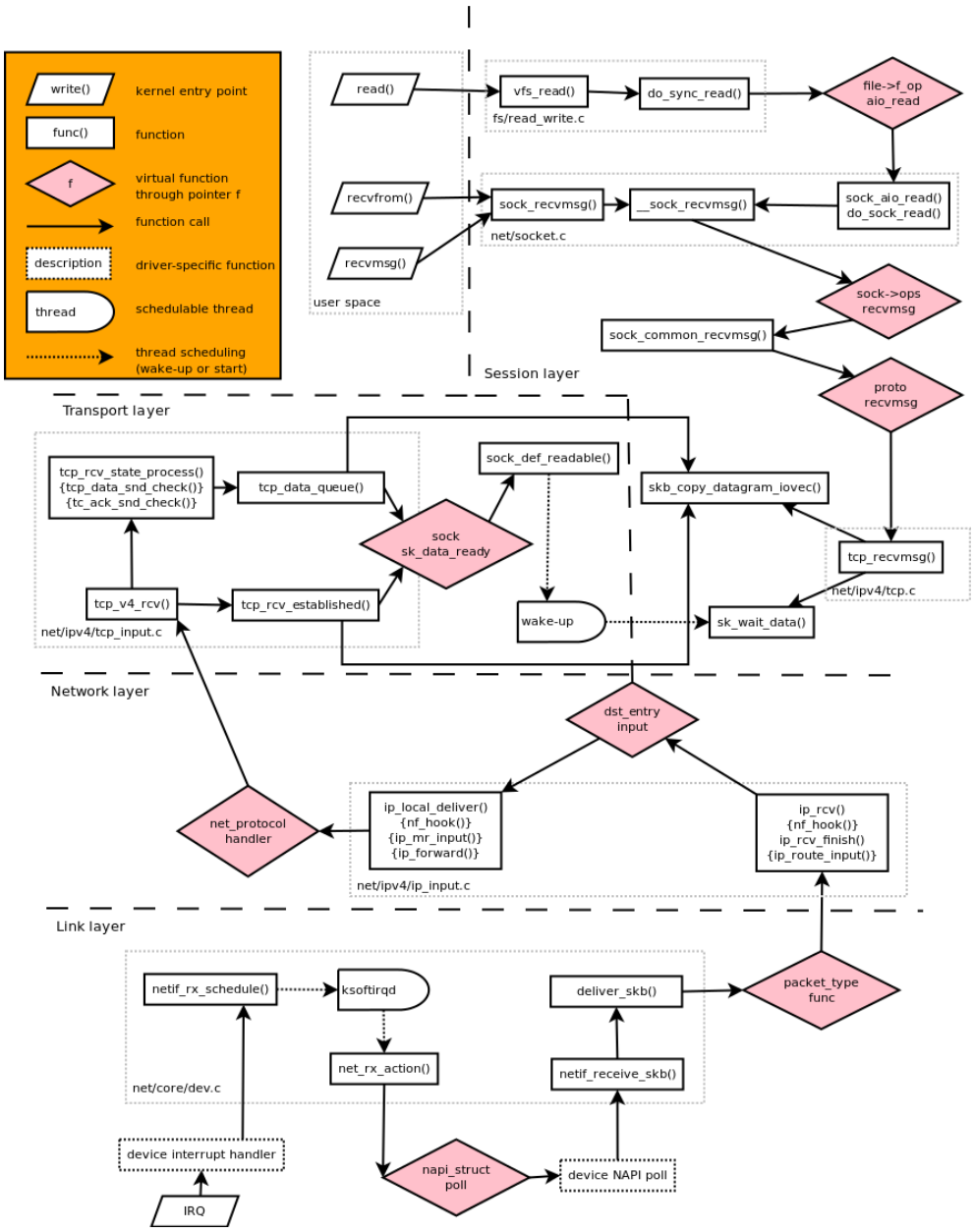


Figure 2.2: Receive flow

- **Layer 3: Network layer (IPv4)**

  The packet will now go to IPv4's packet handler *ip_rcv()*, which will parse the header and check for validity. It will find the destination address by using *ip_route_input()*, and calling the destination's input virtual method. The *ip_mr_input()* method will be used for multicast addresses, *ip_forward()* for packets that have a different destination for which we have a route and *ip_local_deliver()* if the destination of the packet is this machine. This is also the place where datagram fragments will be collected and reassembled. The packet will be delivered to the transport layer by calling the protocol handler for the protocol specified in the datagram through *ip_local_deliver()*.

- **Layer 4: Transport layer (TCP)**

  TCP's protocol handler when using IPv4 is *tcp_v4_rcv()*, which does protocol processing in TCP, like setting up connections, performing flow control, etc. A TCP packet arriving might have an included acknowledgement of a previously sent packet, so this might trigger further sending of packets or acknowledgements, by *tcp_data_snd_check()* and *tcp_ack_snd_check()* respectively. For passing data to an upper layer TCP will use *tcp_rcv_established()* and *tcp_data_queue()*, which maintain the out_of_order_queue of the TCP connection and the sk_receive_queue and sk_async_wait_queue of the socket. The data will be added to one of the socket's queues to wait for the session layer asking for it. If a process is already waiting for data, the data will instead be copied to user space immediately by calling *skb_copy_datagram_iovec()*. The receive functions will call the socket's sk_data_ready virtual method and signal that new data is available and wake up the waiting process.

- **Layer 5: Session layer (sockets and files)**

  As with sending, the application layer have three system calls it can use to receive data from the network: *read()*, *recvfrom()* and *recvmsg()*. All three will eventually end up in *__sock_recvmsg()*. This method will check permissions with *security_sock_recvmsg()*, before it tries to request data from the lower layer using the socket's recvmsg virtual method. *sock_common_recvmsg()* is usually the one, which in turn calls the socket's protocol's virtual recvmsg method. In TCP this is *tcp_recvmsg()* which will either copy the data with *skb_copy_datagram_iovec()*, or wait for it to arrive with *sk_wait_data()*. *sk_wait_data()* is a blocking call that will be woken up by transport layer processing.

2.3 Traffic control

On a router, the set of queueing systems and mechanisms for how the router receives and transmits packets are called traffic control. This consists of how it accepts packets and at what rate on the input of an interface and which packets it transmits in what order and at what rate on the output of an interface. In most cases traffic control is just a simple scheme consisting of one single queue which dequeues packets as fast as the underlying device can accept them. This is a simple first-in first-out (FIFO) queue where the first packet that gets accepted by the queue is also the first packet that leaves.

Packet-switched networks, like IP networks, groups all data into packets and are usually shared between many computers. There is a high probability of many sources sending data at the same time, and a packet-switched network is therefore designed to route each packet independently, with the goals to optimize network utilization, minimize response times and increasing the robustness of the ongoing communication. A packet-switched network is stateless, as each packet is treated as an independent transaction that is unrelated to all previous communication. This statelessness creates the weakness that there is no differentiation between different flows of traffic. Traffic control can be used to mitigate this weakness as it can be used to queue packets differently based on the characteristics of a packet to maximize the usability of the network connection. Following is a small list of possibilities that traffic control introduces:

- Prioritizing latency sensitive traffic
- Limiting or reserving bandwidth of a specific user, service or application
- Dropping a particular type of traffic

Using traffic control can limit the competition for network resources and lead to a more predictable use of these resources. Bulk download traffic can be given a good amount of bandwidth while still servicing traffic with higher priority. For example, if a network has a high download usage through torrent downloading and a user wants to use some highly interactive traffic like voice over IP (VoIP), the VoIP experience will probably be quite bad on an uncontrolled network. Bulk downloading will quickly fill up the queues in the network and packets will get lost or delayed. A lost or delayed packet in VoIP means that parts of the conversation gets lost or delayed. Correctly configured traffic control can give the VoIP packets priority and leads to a smoother VoIP experience at very little cost to the bulk download. Below is a list of the important terms of traffic control:

- A *queue* is a finite buffer where it is possible to line up jobs for a computer or device. In networking, a queue is used to place packets waiting to be transmitted by a device. A packet entering a queue is *enqueued*, while a packet leaving the queue is *dequeued*. The basic scenario consist of a FIFO queue, but it can become much more complex by using other types of queues that can do packet delays, packet prioritization, or even using sub-queues.

- A *flow* in computer networking can be defined as a sequence of packets from a source to a destination. In TCP, a connection with a source IP and port and a destination IP and port will represent a flow. Many traffic control mechanisms work on flows, trying to divide bandwidth equally between competing flows, or prioritizing an interactive flow over a bulk downloading one.

- *Shaping* can be used to optimize or guarantee performance over a network connection. According to Blake, et al. [3]: *"Shapers delay some or all of the packets in a traffic stream in order to bring the stream into compliance with a traffic profile"*. This is the basic idea of traffic shaping; delaying some or all packets to make them conform to a desired rate. One of the big advantages of shaping bandwidth is the ability to control latency. If a link becomes heavily saturated the queues are going to fill up and the latency will rise. By controlling the quantity of traffic being sent out on the network, it is possible to hinder this saturation to happen and the latency will stay low. An implementation with tokens and buckets is much used to achieve traffic shaping:

  - *Tokens* and *buckets* are two related concepts when it comes to traffic shaping. The *token bucket* algorithm is based on a fixed capacity bucket where tokens are added at a fixed rate. The tokens usually represents a single packet or a unit of bytes with a configured maximum size. When a packet is to be sent the algorithm checks the bucket to see if there are enough tokens to send it. If there are the amount of tokens needed for the packet, for example one if it uses packet based tokens, it can be sent out and the token/tokens needed are removed from the bucket. If there is not enough tokens the packet can be treated in different ways, it might have to wait for tokens to become available, or it might just be dropped. The depth of the bucket determines how many tokens can be "saved up" and when the bucket is full it will not collect any more tokens. These saved up tokens will be available at once the the packets arriving and will result in a burst of packets being released to the network based on the bucket size. A flow can therefore send traffic at an average rate

up to the rate of which tokens are added to the bucket, and can have a burstiness as much as the depth of the bucket.

All traffic shapers will have a finite buffer, and must therefore be able to handle the case of a full buffer. The simple and usual way of doing this is to just drop all packets that arrive while the buffer is full. Other more advanced schemes for dropping might be used such as random early detection (RED), which is explained in section 2.3.3. Or a crude alternative would be to allow traffic that exceeds the configured rate to pass through unshaped.

- *Policing* is a technique used to control the rate of traffic that is sent or received on a network interface. Policers or droppers will measure and limit traffic in a network. According to Blake, et al. [3]: *"Droppers discard some or all of the packets in a traffic stream in order to bring the stream into compliance with a traffic profile. This process is know as "policing" the stream"*. Traffic gets measured by the policer and it will take action according to configuration depending on whether the traffic exceeds the profile or not. Possible actions consist of discarding the packet immediately, marking it as non-compliant, reclassifying it or leaving it as it is.

- *Scheduling* in computer science is a method to give give access to system resources, in networking this would be to give packet flows access to the network. The scheduler is concerned with throughput, latency and fairness. These goals often conflict and the scheduling algorithm will usually implement a compromise between these, there exist schedulers of course that has a specific goal, and thus prioritise some parameters over others. Since a scheduler arranges and rearranges packets between input and output of a queue, almost all traffic control mechanisms on a queue can be viewed as a scheduler.

- *Classifying* a packet consists of determining which flow a packet belongs to based on one or more fields in the packet header [16]. A packet classifier consist of a set of rules or policies. Each of these rules specifies what class a packet may belong to based on the fields in its header. This will be used to separate the packets for different treatment as an action will be taken for each class such as marking the packet, routing decisions or packet scheduling in an output queue.

- *Marking* is a mechanism for altering a packet. This could, for example, be to let the underlying network know that this packet is to be prioritized for low latency.

- *Dropping* is a mechanism for discarding an entire packet, flow or classification. For example, a packet that wants to enter a queue that is already full would probably get discarded as there is no more room to store it. The usual way to drop packets is with a *tail drop* scheme. This is the traditional way of dealing with full buffers. When a packet arrives when there is no more room in the queue it will just be dropped, resulting in *tail* drops. Another dropping scheme is *head drop*, which works in the same way as tail drop except the new packet gets accepted after the queue has dropped the first packet. This means that the *oldest* packet will be dropped first, which might be preferable in situations such as in sensor networks where the *newer* packets represents more useful information. Also, if the queueing delay is large, dropping at the head of the line will signal loss to TCP streams faster than dropping at the end. Resulting in TCP streams reducing their rate quicker than with tail-dropping.

- *Active queue management*, or just *AQM*, is a term used for all schemes that uses a proactive approach to queue management [4]. This usually consist of dropping or marking packets before buffers overflow. They typically work by probabilistically dropping or marking packets with a higher probability the fuller the queue gets, but there are of course other schemes too. With tail drop queues there will be no indication to the endpoints that the network is overflowing before a queue is actually full. AQM will provide endpoints with an indication that the network is starting to get congested before it is full. Which can lead to congestion avoidance, if the transport protocol is using congestion control. This has the added benefit of shorter queues that will reduce network latency. A major drawback of AQM is that they are usually very hard to configure right. The wrong configuration might lead to worse conditions than just a congested network. An example of an AQM scheme is RED, explained in section 2.3.3.

While traffic control sounds great, there are some disadvantages. The first and most important is the complexity of configuring traffic control. While traffic control used correctly can give a better distribution of network resources, it could just as easily be used incorrectly and introduce more contention for the same network resources. This could lead to worse utilization of the available resources than not using traffic control at all. Another small disadvantage is the increased computing resources needed to handle the increased processing of packet routing in a traffic control scenario.

2.3.1 Traffic control in Linux

Linux provides the *iproute2* collection [9], which includes several tools to configuring and controlling the TCP/IP networking and traffic control. It provides several tools, but the most noticeable among them is the *ip* and *tc* tools. The *ip* tool is used to configure IPv4 and IPv6, and can be used to do a host of different things including configuration of a network device and managing the neighbour and routing tables. The *tc* tool is used to configure the central elements in Linux traffic control, the queueing disciplines and filters, which will be explained in detail in the following sections.

2.3.2 Queueing disciplines in Linux

A queueing discipline (qdisc) can be looked at as a scheduler or an algorithm that manages the queue of a device [18]. It is important to understand that queueing is used to determine the way we *send* data. This is because with the way networks are designed we have no way to control what other computers send us. Queueing disciplines govern how packets are buffered when waiting to be transmitted and can make decisions on which packets to send based on policy settings.

In Linux a queueing discipline is implemented as a simple object with two key interfaces. One interface is for queueing packets to be sent while the other is for releasing packets to the network device. This is the building block on which all of Linux traffic control is built. The architecture of queueing disciplines in Linux is shown by figure 2.3. A queueing discipline exists in the link layer, between the layer 3 protocol and the network device. There exists a rich set of queueing disciplines in Linux for doing advanced queueing, prioritization, and rate control. The queueing discipline attached to a device is called a root qdisc, and all network devices have exactly one root qdisc.

Queueing disciplines can be split into two groups. Classless queueing disciplines and classful queueing disciplines. A classful qdisc can contain multiple classes, all of which are internal to the qdisc, and provides a handle that can be used to attach filters.

A class is very flexible and can contain ether several children classes or a single child qdisc. A class can also contain a classful qdisc, which means that it is possible to create increasingly complex traffic control scenarios. A leaf class is a class that can have no child classes, and will contain exactly one qdisc. This will be a qdisc that is responsible for sending data from the class. When a class is created in Linux it will have a default fifo qdisc attached to it and is by definition a leaf class. When a child is added, the qdisc attached is removed, and it is then called an inner class.

A filter can be used to classify packets. It will contain a variable number of conditions that can be used to match packets to the filter and

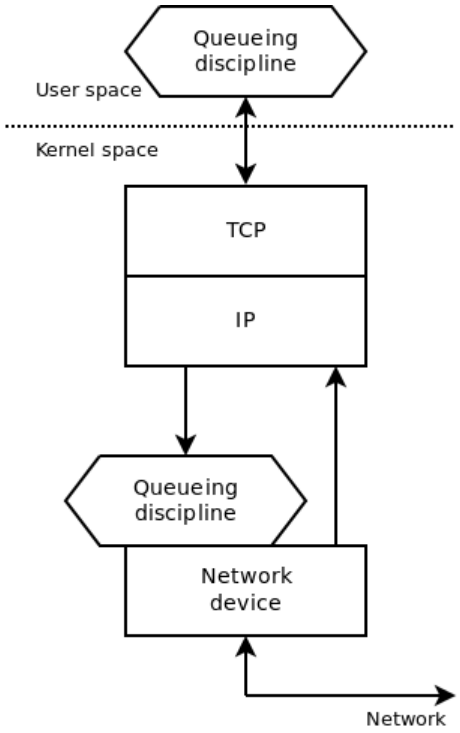


Figure 2.3: Linux queueing discipline

send it to the right qdisc. It is possible to attach filters to classes or to classful qdiscs. In a hierarchy of classes and qdiscs, the packet will always be matched first with the filter attached to the root qdisc. It can then be directed to any subclasses where the packet may be enqueued at a qdisc or classified again.

A classless qdisc is a qdisc that can have no children of any kind and it has no configurable internal subdivisions. This means that the qdisc contains no classes and as there is no reason to classify packets it will neither contain any filters. A typical example of a qdisc considered classless is the qdisc used by default under Linux, *pfifo_fast*. An interesting detail about the pfifo_fast qdisc is that it has three bands that is used to prioritize traffic. *pfifo_fast* and other queuing disciplines will be explained in the following sections.

Another attribute of a queueing discipline is whether it is Non- or Work-conserving. A work-conserving queueing discipline will always deliver a packet if it has one available. This means that it will never delay a packet if the network device is ready to send one. A non-work-conserving queueing discipline can hold a packet even if the device asks for one. This is for example done to limit the bandwidth, the case with TBF, or delay a packet for emulation, like done in NetEm.

2.3.3 Classless queueing disciplines

A classless queueing discipline contains no classes and can have no filters attached. An explanation of the classless pfifo/bfifo, pfifo_fast and RED queueing disciplines will be given in this section

FIFO and pfifo_fast

The First-In First-Out (FIFO [35]) qdisc is the most basic of all queueing disciplines and the one used by default inside all new classes. A FIFO queue will keep a list with all packets. All packets added to the qdisc will be placed at the tail of the list, while all packets that will be sent from the qdisc will be taken from the head. This means that, as the name states, the first packet enqueued in the qdisc is also the first packet that will be dequeued. It will perform no shaping or rearranging of packets, and it will transmit the packets as fast as the underlying device can handle. If the list of packets, also known as the *buffer*, grows to a configured max size it will start dropping all packets trying to enqueue until there is room. There are three FIFO qdisc variants with small differences, *pfifo*, *bfifo* and *pfifo_head_drop*. The difference between pfifo and bfifo is in how they handle buffering. The buffer size in pfifo is measured in packets, while the buffer size in bfifo is in bytes. All packets will therefore be equal from a pfifo perspective as all packets occupy one slot in the buffer. In bfifo a nearly full queue might drop a large packet but allow a small one if it fits in the space left. Pfifo_head_drop is just a pfifo queue where all dropping happens at the head of the queue.

The *pfifo_fast* qdisc is the default qdisc of each interface in Linux [39]. This qdisc is based on FIFO, but can offer some prioritization. It has three internal bands, which are all FIFO queues. Packets arriving at the qdisc will be enqueued into one of these queues based on their priority. Packets waiting in band 1 will be sent out before packets in band 2, and similarly packets in band 2 before packets in band 3. The classifier in pfifo_fast uses the Type of Service (TOS) bits in the IP header to determine which queue a packet belongs to based on the default Linux priomap. The priomap in pfifo_fast is the default Linux priomap and cant be changed. There are 5 types of traffic defined with the TOS bits: minimize delay, maximize throughput, maximize reliability, minimize monetary cost and normal service. There is also the possibility of combining these. This will be used to prioritize the packets with high priority such as interactive packets. As these are FIFO queues, all packets within a class will be sent in the order they arrived. It is worth noting these bands are not real classes and the FIFO queue can therefore not be changes to any other queueing scheme. A simple representation of the pfifo_fast qdisc can be seen in figure 2.4. In this example the assumption is taken that none of the 4 packets arriving are dequeued before they are all inside the qdisc. The

red packet is the third to arrive, but with its high priority it is place in class 1 and therefore dequeued before the other three packets that arrived. The yellow packet, first to arrive, which had the least priority is placed in class 3 and is dequeued after class 1 and 2 is empty.

Figure 2.4: pfifo_fast has three bands that can be used to prioritize traffic

A possible problem with pfifo_fast is the presence of a large amount of high priority traffic. The pfifo_fast qdisc will only dequeue packets from a lower priority band as long as the higher priority bands are empty. A full high priority band can lead to starvation of the lower bands, in the worst case it lead to non-interactive low priority flows not getting any traffic through at all.

RED

Random Early Detection (RED [41]) is a classless qdisc that uses a variable probability for dropping packets to manage its queue size. It is designed to be used in a network where a single marked or dropped packet is enough to help a transport layer protocol notice congestion [8]. The variable drop probability will result in the queue dropping packets before it is full and consequently signal a congestion control mechanism that there is an impending link congestion causing flows to slow down before the queue is full. This will, of course, not help much if a flow has no congestion control, but RED can still have some use in the presence of no congestion control. With a regular tail drop algorithm, a queue will simply be filled until it is full, and drop all packets that arrive while it has no more space available. This means tail drop will distribute buffer space unfairly among the flows. A flow which uses a high amount of bandwidth will have a higher chance of taking the space that is free in the queue as it has more packets arriving. In the case of RED, which does its dropping with a probability, this means that the more a host transmits the larger the chance that one of its packets

will be dropped. This is also the reason why RED still has some uses even in networks without congestion control. The goal of RED is to have a small average queue size, which is good for interactive traffic while not disturbing congestion controlled traffic with too many sudden drops after a burst in traffic.

The RED algorithm uses the average queue size to calculate the dropping probability. This average is compared to two configurable parameters: the *minimum* and the *maximum* threshold. While the average queue size is less than mimimum, RED will enqueue all packets arriving. As soon as the average size exceeds minimum it will start dropping packets. The probability of doing this will rise linearly up to a configured maximum *probability*. It will reach the maximum probability when the average queue size equals the maximum threshold. As the maximum probability is not normally set to 100%, there is a possibility of the queue size growing to a larger size than the maximum threshold. In this case there is also a hard limit on the queue size that if reached will make the RED queue behave like a tail drop as there is no more buffer space to store packets.

RED also has the possibility of marking packets instead of just dropping them outright. In Linux marking can be achieved by configuring RED with Explicit Congestion Notification (ECN [31]). ECN will allow RED to set a mark in the IP header of the packet notifying the receiver of the packet to signal the impending congestion in the network. The receiver of the packet with the ECN mark will send the congestion indicator back to the sender which will then reduce its transmission rate to avoid full congestion in the network. Both sender and receiver must agree to use ECN or the marking will have no effect, so RED will drop, instead of mark, all packets from non-ECN flows.

2.3.4 Classful queueing disciplines

A classful queueing discipline can have filters and can contain subclasses. This makes classful queueing disciplines very useful to treat traffic in different ways, giving one type of traffic priority over another. The root of an interface will have one root qdisc. The default pfifo_fast qdisc might be swapped out for a classful one, for example HTB. To this there might be attached more qdiscs and so forth. This can result in a class based set-up, e.g. the one shown in figure 2.5.

An important thing to know is that the surrounding networking system only knows about the root qdisc, which is where all the queuing and dequeuing calls are made. In addition, only a leaf qdisc can hold queue packets. This means that a packet being enqueued into a system with multiple qdiscs will "flow" towards the leaf qdiscs, while packets being dequeued will "flow" back up towards the root. In the example

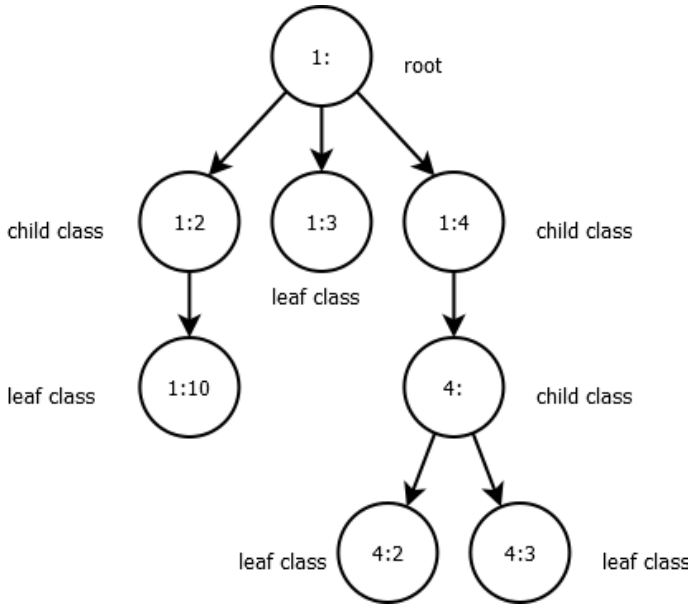


Figure 2.5: Class example

mentioned above packets can only be stored in 1:10, 1:3, 4:2 and 4:3, as these are the leaf classes. When the system wants to enqueue a packet it must start at the root. As this is not a leaf qdisc it will try to enqueue the packet in one of its child qdiscs. If this is not a leaf qdisc either it will again ask its children and so forth, until the packet gets enqueued in a leaf class. When the system wants to dequeue a packet it will also start at the root qdisc. Since this qdisc can store no packets it will ask one of its children. If this qdisc has no packets, it must ask its children. When a qdisc has a packet that can be dequeued it will be returned through the qdiscs back to the root, which will return it to the networking stack.

As examples of classful queueing disciplines, the PRIO, TBF and HTB qdiscs will be explained.

PRIO

The priority qdisc (PRIO [40]) is a simple classful queueing discipline that can have a configurable number of classes with different priority. It will perform no shaping and only divide traffic based on its configuration. By default PRIO will have 3 bands, each containing a FIFO qdisc, and behave much like pfifo_fast. Packets are prioritized according to the TOS bits and the default priomap, and sent to the corresponding class. These classes are dequeued in a descending order of priority. What makes PRIO different from pfifo_fast is its configuration possibilities. The pfifo_fast qdisc can only divide traffic into its bands by the TOS field in the packet header, whereas PRIO has three different ways to do the same. The first possibility is the same as in pfifo_fast, dividing traffic based on the TOS field and a

configurable priomap. The second is the possibility that a process with high enough privileges can encode the destination class in userspace. The final way it can divide traffic is possible by PRIO being a classful qdisc. A filter can be attached to provide classification as explained in section 2.3.5.

As PRIO behaves much like pfifo_fast in that it dequeues from the higher priority bands before the lower, the same problem of starvation persists here. But in PRIO there is a way to minimize or even remove this problem. As each band is a class, it is possible to swap out a FIFO queue with some other queueing discipline. This makes it possible to change the default FIFO qdisc for a qdisc which can do rate limitation. A rate limitation qdisc on the higher priority bands can make sure that the higher priority traffic does not starve all other flows.

In PRIO there is also the possibility at creation time to configure the amount of bands available to further fine-tune the prioritization of traffic flows. It is worth noting that the priomap must be configured if more than three bands are to be used. This is because the default priomap only divides traffic into three bands.

TBF

Token Bucket Filter (TBF [42]) is a classful queueing discipline which uses the token bucket algorithm to shape traffic. When a TBF qdisc is applied to an interface its bucket is full with tokens, this is the maximum of traffic that can be sent out on the network in one burst. This bucket is constantly filled with tokens at a specific rate until the bucket is full. All packets enqueued at a TBF qdisc needs a token to be dequeued and when it is, the token used will be deleted from the bucket. Figure 2.6 illustrates the two flows used in this algorithm: the token flow and the data flow. These two flows gives us three possible scenarios that can happen in a TBF queue. The first scenario is when the data that arrives in the TBF qdisc is at a rate that is less than the rate at which tokens arrive. This will mean that the data will not use all the tokens created and the bucket will begin to fill up until the bucket is full. These tokens can then be used to send data out faster than the configured rate and will lead to short bursts of data. The second scenario is when data arrives in the TBF qdisc at the same rate as tokens. In this case all packets will be sent out with no delay. The third scenario comes into effect when the packets arrives in TBF at a rate that is greater than the token creation rate. The bucket will run out of tokens fast and data will start filling up the queue. When the queue is full all other arriving packets will be dropped. In the TBF implementation in Linux tokens do not represent packets, instead tokens correspond to bytes. This means that packets will need several tokens in order to be dequeued.

TBF is, as mentioned, a classful qdisc, and when a TBF qdisc is created, a class with a basic bfifo queue is attached. This queue is used as the data queue and can be changed to any of the other qdiscs available. TBF has

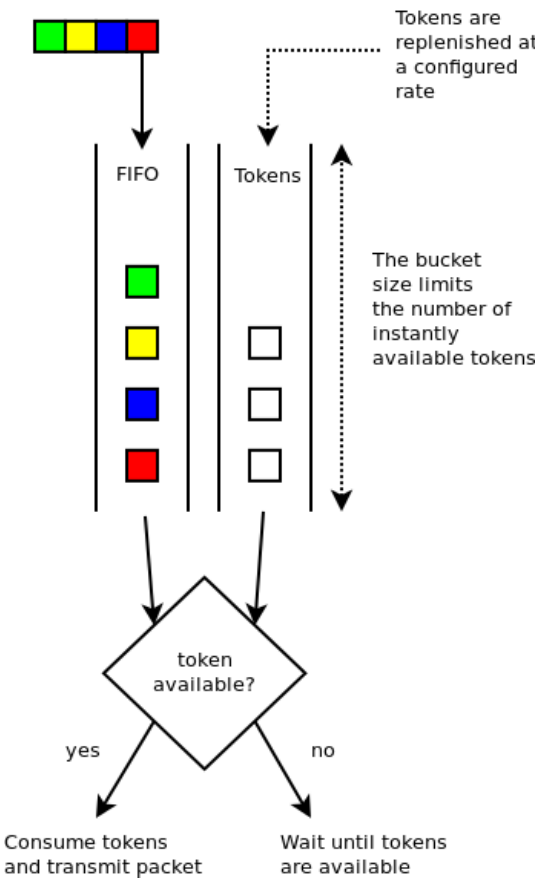


Figure 2.6: TBF has one queue and a bucket for tokens

many possibilities for fine tuned configuration which will be explained below:

- **limit:** This parameter will limit the maximum amount of bytes that can wait in the bfifo queue for tokens to become available. It is mutually exclusive with the latency parameter.
- **latency:** The latency parameter can be used to specify the maximum amount of time a packet can remain in the queue waiting for a token. It will calculate the queue size based on the size of the bucket, the rate at which tokens is spawned and can also take into account the peakrate.
- **burst:** Also called buffer or maxburst. This will set the size of the bucket in bytes. This equals to the maximum amount of bytes that can be dequeued instantly. A larger rate will need a larger buffer to reach the desired rate.
- **mpu:** This is the minimum token usage in bytes a packet can use. Even a zero-sized packet uses some bandwidth. For ethernet the minimum size of a packet is 64 bytes. The mpu parameter can be used to represent this.
- **rate:** The rate parameter is used to set the desired rate. This will determine how fast tokens are generated.
- **peakrate:** By default, if there are tokens available when a burst of traffic arrives at the qdisc it can be sent out immediately in a burst that might exceed the wanted rate by a large amount, especially if the bucket is large. The peakrate parameter is used to limit how fast a bucket can be emptied and even prevent bursting over the desired rate entirely. This will create a second bucket with a size of one packet, which will mean that there can be no burst higher than the peakrate.
- **mtu:** This parameter can be used together with peakrate to set the size of the peakrate bucket in bytes. This can be used if peakrate is needed, but it is still acceptable to have some burst.

HTB

Hierarchical Token Bucket (HTB [36]) is a classful queueing discipline that can be used to shape traffic using the token bucket algorithm [25]. It has the possibility to do prioritization, guarantee minimum rates, and even reallocate excess bandwidth using a complex borrowing algorithm. HTB is used to control the outgoing bandwidth on one physical link to simulate several slower links and sending different traffic on the different simulated

links. To manage this HTB uses classes and filters. HTB's borrowing algorithm combined with an arbitrary amount of classes and filters can be used to control traffic with a very fine-grained precision. The fact that child classes can borrow tokens from parent classes makes it possible to use unused bandwidth from other child classes. This way all child classes can get its minimum guaranteed bandwidth without affecting other classes, but still use more bandwidth if some of the other classes are not using theirs. This makes HTB extremely versatile in dividing bandwidth between flows with a guaranteed minimum and still keep a high link utilization.

An arbitrary amount of classes may exist in HTB, with all of these classes containing a default pfifo qdisc. Figure 2.7 shows an example of a class hierarchy that can be used with HTB. There are two types of classes, inner nodes and leaf nodes. All classes that do not contain a child is a leaf node, while all other classes are inner nodes. Only a leaf node can hold a packet queue. When a packet is enqueued in HTB it will start at the root

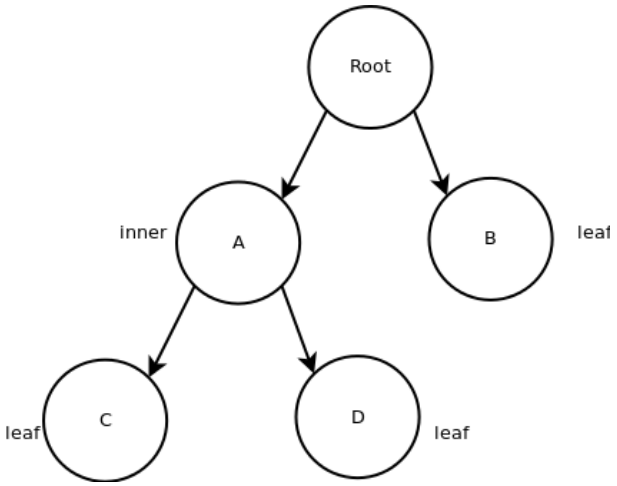


Figure 2.7: HTB hierarchy

node. At each node HTB will check all filters attached to see where it is supposed to redirect the packet. If it is redirected to a leaf class, it will be enqueued to the attached qdisc. If it is redirected to another class, then the filters attached to this class is checked to find out where it is supposed to go next. For example, when a packet is enqueued in the example given in figure 2.7, the filters attached to the root qdisc is examined. Then it is sent to class A and since this is an inner class, A's attached filters will now be checked. It is sent to D where it is enqueued at the attached default pfifo qdisc as this is a leaf node.

The borrowing algorithm is a core part of HTB and what makes link sharing and reallocation of bandwidth possible. Each class will have an *assured rate* (AR) and a *ceil rate* (CR). The assured rate is the guaranteed minimum rate a class will have available at all times and the ceil rate is the

absolute maximum a class can use. The actual rate a class uses is given by R. While a leaf class has an R less than AR it will dequeue packets as long as there are available tokens. If it exceeds AR it will try to borrow tokens from its parent class as long as it does not reach CR. As long as a child class has an R greater than or equal to CR it will not be able to dequeue any packets. While an inner class has an R less than AR it will lend token to its children. When it reaches AR it will try to borrow tokens from its parent, that it can lend to its children, as long as it does not reach CR. If an inner class reaches CR it will neither lend nor borrow tokens until its R is less than CR. Shaping is only done in the leaf classes, therefore the sum of the assured rates of the children of a particular class should not exceed the ceil rate of their parent class. To get the best utilization of resources the sum of children rates should match the parent assured rate, this will allow the parent to distribute the ceil rate bandwidth between the children classes. Distribution of leftover bandwidth is done in a round-robin fashion. All classes must be given a priority and the class with the highest priority will get available bandwidth first.

Based on the borrowing algorithm all classes will at any given time have one of three modes that is computed from R, AR and CR. The possible modes are red, yellow and green and corresponds to what a class can do. Red means a class cant send as it is over its assigned ceil rate. Yellow means a class can borrow, it is over its assured rate but has not yet reached its ceil, and green means the class can send as it is still under its assured rate.

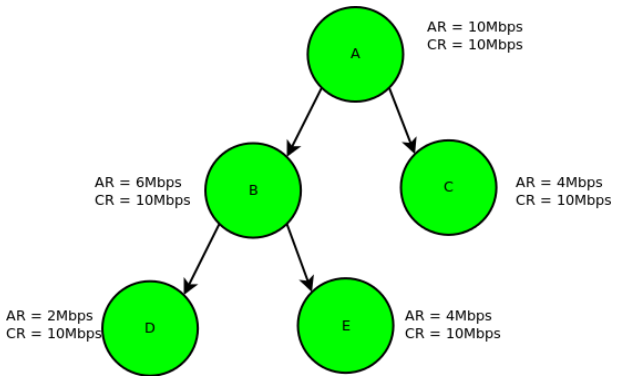


Figure 2.8: Without traffic

Figure 2.8 shows a simple HTB set-up with no traffic. A root node named A, an inner node named B and three leaf nodes named C, D and E. Class A has an assured rate equal to its ceil rate of 10Mbps. As this is the root node this will be the maximum rate for the entire configuration. Class A has two child classes, both of which has a ceil rate of 10Mbps. B has an assured rate of 6Mbps, while C has 4Mbps. D and E both have a ceil equal to their parent's ceil of 10Mbps, and an assured rate of 2Mbps and 4Mbps respectively. This set-up makes it possible for B and C to borrow from A,

and B to lend bandwidth to D and E.

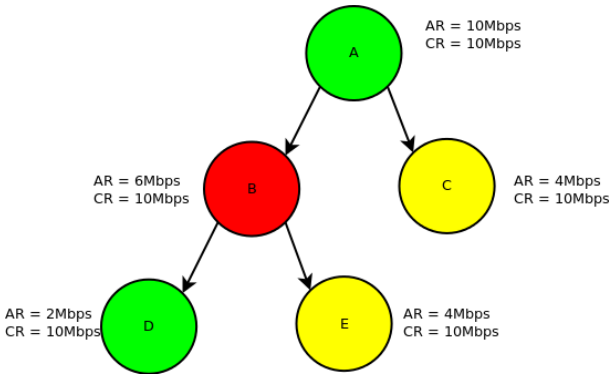


Figure 2.9: C and E are over their rate, B is over its ceil

A possible example state of HTB with traffic flowing through is shown in figure 2.9. B is over its ceil, which means that it can neither lend tokens to its children or borrow tokens from its parent. C is over its rate, but can still try to borrow tokens from A. E is also over its rate so it has to borrow tokens from its parent, but since B is over its ceil it cannot get any and have to wait for tokens to become available. Therefore it is only possible for C and D to dequeue packets in this scenario.

2.3.5 Packet classification with filters

A classful queueing discipline will often consist of several classes arranged in a treelike structure. If a packet arrives at a class with subclasses, it will need to be classified to find out which class it should go to. Two of the main ways of doing packet classification in Linux is with iptables [26] or with filters configured in tc. When a packet arrives at a class that needs to do classification all filters attached to it will be checked to decide where the packet should go. The filters will be checked at each class until it reaches a leaf class and is enqueued. Consider figure 2.5, an example tree of a classful queueing scheme. When a packet enters the qdisc it will always check filters at the root first. From there the packet might be sent to class 1:4. Here the filters attached to this class must be checked and it might be sent to class 4:. Based on its filters it might end up in 4:2 where it will be enqueued. Filtering may never flow upwards, so if a packet reaches level 2 in the tree it may never go back up to level 1 or 0. It is possible however to have filters that direct packets past levels. For example, a filter attached to the root might send a specified type of traffic directly to class 4:2.

The tc tool provides several different filters of which the most used are explained below:

- **fw:** This type of filter can be used together with classification by the iptables tool. Instead of iptables doing classification it can be

configure to mark packets. This filter will do classification based on these marks.

- **u32:** The u32 filter type is used to match on any part of the packet. For example, u32 can be used to match source or destination addresses or ports. It can be used to match the protocol used, or the TOS field in an IP packet. As any part of the packet can be matched there is a possibility for really complex matching schemes.

- **route:** Another filter type is the route classifier filters. This filter can classify based on defined flows as defined in the routing table. This gives the possibility of classifying traffic based where it is from, where it is going to or which interface the traffic arrived on.

2.3.6 Ingress qdisc

In linux, queueing disciplines are attached to a network interface and everything that is queued to the interface is queued to the qdisc first. Consequently all the qdiscs that has been explained so far in this chapter is for outbound traffic, also called egress qdiscs. There is a possibility to attach a special qdisc to interface for ingress traffic aptly named the *ingress* qdisc [37]. There is very little that can be done with the ingress qdisc. It contains no queue or classes and only serve as a point to attach filters. This makes it possible to police incoming traffic, before the traffic even enters the network layer, as tc filters contain a full token bucket implementation. No delay can be introduced here as there is no packet queue and packets will either be dropped or passed on. There are workarounds for using egress qdiscs on ingress traffic such as the Linux Intermediate Queueing Device (IMQ [23]), but that is beyond the scope of this thesis.

2.4 NetEm

NetEm is a network emulator included in the Linux kernel presented in [17]. It focuses on providing emulation functionality that can be used to test protocols and applications. It gives the user a possibility to emulate such things as packet loss, duplication, reordering and packet corruption. NetEm can also be used to introduce packet delay and add random jitter.

NetEm has seen some changes throughout the years it has been available. It started out as a classful qdisc, before the classful functionality was removed in 2008 because of problems with the dequeue/requeue interface. This removal was reverted in 2011, after the qdisc API was reworked, to bring back the possibility to change the inner queue used by NetEm. It also had no built in rate control, which meant it had to rely on other qdiscs for this. In late 2011 a built in rate control was added to

NetEm, but it had some problems with delay calculation that made it not usable. A fix for this was introduced in early 2013, and it is now working properly.

2.4.1 Design

The NetEm emulator has in two parts. The first part is a small kernel module which consists of a queueing discipline and has been integrated as a part of the Linux kernel since version 2.6.8. The second is a command line utility to configure it called tc. Tc, described in 2.3.1, is a traffic control tool which is a part of the iproute2 [9] package, and contains the command line utility to configure the queueing discipline.

To the surrounding system all queueing disciplines work in the same manner, and are placed in the link layer between the protocol output in the network layer and the network device, as shown in figure 2.3. NetEm will have packets enqueued and dequeued in the samme manner as all other queueing disciplines. Figure 2.1 presented earlier gives a quick overview of where the system enqueues and dequeues packets from the qdisc.

Emulation by the NetEm qdisc is done at enqueue time. All packets being enqueued to the qdisc will be subjected to the configuration parameters, getting their *time_to_send* calculated and placed in an internal tfifo (time fifo) queue. This is a modified FIFO queue where all packets are arranged based on the time_to_send time stamp. In most cases this means that the packet will be placed atthe end of the queue, but special cases such as jitter or delay distribution might place packets earlier in the queue. The time_to_send value of a packet is calculated based on what type of emulation being done, and once it is placed in the tfifo queue it is subject to no more emulation.

When a packet is enqueued at a qdisc by the kernel it will also try to dequeue a packet. As the dequeue part is called by the kernel, NetEm will check its queue to see if there is a packet to send based on the time set when it was enqueued. If there is, the packet is dequeued. If there are packets that cannot be sent yet, then the dequeue function will set a timer that will fire when packet is ready.

The general tc command to configure netem is given below:

`tc qdisc` *`action`* `dev` *`interface`* `root netem` *`options`*

A NetEm qdisc will be created or changed, according to *action*, *add* and *change* respectively. The qdisc will have the configuration given by *options* on the interface given by *interface*.

2.4.2 Packet delay

The delay parameters in NetEm are described by an average value (μ), standard deviation (σ), and correlation (ρ). The average value can be

specified in milliseconds and will cause NetEm to delay all packets by this amount of time. An example command that would add 100 milliseconds static delay to all packets going through the qdisc would be:

```
tc qdisc add dev eth0 root netem delay 100ms
```

As real wide area networks do not have a constant delay and will show variability, it is possible to specify an optional standard deviation in milliseconds to add a random variation to the constant delay. For example, if 100 ms is chosen as the average value with a random variation of 20 ms then NetEm will schedule all packets to have a random delay between 80 ms and 120 ms. This could be achieved by using the following command:

```
tc qdisc add dev eth0 root netem delay 100ms 20ms
```

As network delay variation is not purely random, NetEm also allows for an optional correlation value specified in percent. A 20% correlation introduced to the last example will make NetEm schedule packets with a random delay between 80 ms and 120 ms where each random element depends 20% on the last one. This command adds correlation to the delay

```
tc qdisc add dev eth0 root netem delay 100ms 20ms 20%
```

NetEm uses by default a uniform distribution ($\mu \pm \sigma$), but it is possible to specify a non-uniform distribution. This random distribution can be derived from a table that is generated either from experimental data such as ping times or from a mathematical model. The command to configure netem with a normal delay distribution is:

```
tc qdisc add dev eth0 root netem delay 100ms 20ms \
                                 distribution normal
```

The iproute2 package includes tools to generate a normal distribution, Pareto distribution, Pareto normal distribution and a sample based on experimental data. According to [11], the actual tables (normal, pareto, paretonormal) are generated as part of the iproute2 package and placed in the /usr/lib/tc directory. It is also possible with a bit of work to make new distribution tables based on experimental data to use with NetEm.

2.4.3 Loss, duplication and corruption

Packet loss can be specified in the command line utility as a percentage of packets to be dropped, in addition an optional correlation value also specified in percent may be used as can be seen by the commands below:

```
tc qdisc add dev eth0 root netem loss 0.5%
tc qdisc add dev eth0 root netem loss 0.5% 25%
```

NetEm emulates packet loss by randomly dropping the specified percentage of packets before they are queued. The correlation value will make each successive probability depend on the last one, this will make the random number generator behave less random and can be used to emulate bursty packet losses. There exists a problem with loss correlation in NetEm. This was discussed in the NetEm kernel mailing list, and explained in [22]. One fix for this, which has been implemented in NetEm [33], has introduced new correlated loss models that can be used.

Duplication works much like packet loss. It can be specified as a percentage of packets to duplicate and it can also have an optional correlation value:

```
tc qdisc add dev eth0 root netem duplicate 1%
tc qdisc add dev eth0 root netem duplicate 1% 25%
```

Packet duplication is done in NetEm by randomly cloning a percentage of packets before they are placed in the internal queue. As with loss, duplication can have the optional correlation value to make chances of duplication less random and more lifelike.

Packet corruption works much like duplication and loss:

```
tc qdisc add dev eth0 root netem corrupt 1%
tc qdisc add dev eth0 root netem corrupt 1% 25%
```

To create packet corruption NetEm will introduce a single bit error at a random offset in the packet. This emulates random noise, and can be used to emulate noisy links, such as wireless links.

2.4.4 Reordering

It is possible to use two different ways to specify reordering with NetEm. The first is very simple form of reordering while the second is more like real life. NetEm does its reordering by putting a packet that will be reordered at the front of its internal queue to be sent immediately. This can result in the packets being reordered having a shorter delay time than specified with the delay option.

The simple method of doing reordering can be specified by a gap parameter which takes a number N. This method works by using a fixed sequence and will reorder every Nth packet. For example, if 5 is used as a gap, then every 5th packet will be put at the head of the queue, while all other packets will be delayed. To create a NetEm qdisc with gap reordering of every 5th packet and a delay of 20 milliseconds the following command can be used:

```
tc qdisc add dev eth0 root netem gap 5 delay 20ms
```

The more life like method works a bit like loss and duplication. It can be specified by a reorder parameter which uses a percentage and an optional correlation. This method works by reordering a certain percentage of packets with the optional correlation affecting the next packet. For example, specifying reordering with 10%, will make NetEm randomly select 10% of the packets to send immediately while the rest will be delayed. This example combined with a correlation of 25% and a delay of 20 milliseconds could be achieved by:

```
tc qdisc add dev eth0 root netem reorder 10% 25% delay 20ms
```

Another special case might also cause reordering in NetEm. As the internal queue in NetEm keeps packets in order by their time to send, it is possible to have reordering when using random variation on delaying packets. For example, lets have a delay of 100 ms and a random variation of 50 ms. The first packet gets a delay of 100 ms with a random variation of 0 ms, while the second packet gets a delay of 100 ms with a random variation of -50 ms. Here the first packet will be sent after 100 ms while the second will be sent after 50 ms, this will make NetEm send the second packet before the first.

2.4.5 Rate control

For a long time NetEm had no built in rate control. For this it had to rely on one of the other queueing disciplines that performed bandwidth management, such as Token Bucket Filter (TBF) and Hierarchical Token Bucket (HTB). As an example, a setup using TBF could be something like:

```
tc qdisc add dev eth0 root handle 1:0 tbf rate 1mbit \
                                burst 1540 limit 1540
tc qdisc add dev eth0 parent 1:1 handle 10: netem delay 100ms
```

This will create a qdisc system where packets are first queued in the NetEm queue and subjected to network emulation, before it is passed to the tbf queue and subjected to rate control from where it is dequeued and sent out on the interface. TBF has a host of configurable parameters to get the rate control to behave as needed as explained in section 2.3.4.

A problem with the setup above is the combination of several queueing disciplines and the way they work together. An intimate knowledge of how the queuing disciplines work is needed to configure the system in the correct way. The two resulting queues is one of the troubles of configuring the system to the wanted specifications. These unneeded configuration problems are one of the main focuses for this thesis and will be discussed in section 3.1.

As the rate control implemented in Netem has newly been fixed (version 3.8 of the Linux Kernel), this is now a real alternative to the

configuration problems of having multiple queueing disciplines. NetEm will do rate control by calculating the size of the packet and how much time it will take to get the packet out onto a link with a set bandwidth. All other emulation, like delay, will be added to this time and the packet will be placed in the NetEm internal tfifo queue based on its time_to_send time stamp. Configuring a system like the one above with a bandwidth of 1mbps and a delay of 100ms would be done as follows:

```
tc qdisc add dev eth0 root netem rate 1mbit delay 100ms
```

This approach uses only one queue, the internal NetEm queue, which means that the configuration problems mentioned above is gone. It does introduce another small problem though. The combination of rate and delay in the same queue leads to early orphaning of the skb which breaks mechanisms that is based on the amount of skb ownership by a network socket, such as TCP Small Queues. This is one of the issues that is avoided with the two queue design presented in this thesis.

2.4.6 NetEm limitations

Network emulators not running on dedicated hardware systems, will naturally have some limitations. As the main focus of this thesis focuses on the NetEm emulator, an explanation will be given on the main limitations concerning this emulator.

Random number generator

Emulation can often include a random element, such as introducing random jitter to a link. NetEm uses a Pseudo-Random Number Generator (PRNG) to get these random numbers. These numbers are not true random and the generator used will impact the emulation results. The choice of which PRNG to use needs to take into account the speed and quality of the PRNG. There exists cryptographically secure random number generators in the Linux kernel, such as the *get_random_bytes()* function, but these are usually very slow and does not function very well with emulation that might need a lot of random elements.

Network drivers

Network devices in Linux needs a driver. These are not universal and might have different implementations. These drivers will have a transmit ring (a buffer) that can hold references to data that it wants to transmit through the hardware. The drivers themselves can use these transmit rings differently, and have to do some kind of flow control with regards to this buffer. When the system is under high load, the emulator might release bursts of packets that the driver must be able to handle. Testing

Timer Res	Min	Average	Max	stddev
Ref	0.124	0.209	10.258	0.712
100Hz	11.198	33.280	59.952	8.669
250Hz	6.443	15.899	36.141	4.062
1000Hz	5.074	5.778	16.866	1.053
Hrtimer	2.133	2.216	12.265	0.712

Table 2.1: Timer resolution results

done with NetEm [17] revealed several drivers that could not do flow control properly. This issue could lead to a full stop in transmitting packets from the device.

Timers

One of the most important elements when it comes to emulation is the timer granularity. The more precise a timer is, the more precise the emulation can be. As Linux is not a real-time system there will be some constraints when it comes to the timers. For a long time the Linux kernel was limited by its system time tick. This has usually been set to a rate between 100Hz and 1000Hz, with the usual values of 100, 250 and 1000. This system time tick limited the timer the system could use. 100Hz will interrupt 100 times a second (10ms), and 1000Hz will interrupt 1000 times a second (1ms). This means that the timer granularity cannot be more precise than 1ms with the kernel configure to run at 1000Hz. In theory this leads to NetEm not being able to emulate networks shorter than 1ms. In practice this might even too short. Using a 100Hz kernel, Jurgelionis, et al. reached the conclusion in [21] that: "The main result of the study show that NetEm is able to generate constant delay higher than 50ms". The possibility for NetEm to only release packets every 10ms would lead to some packets being sent later than they should be.

Fortunately, this problem has been mitigated with the implementation of the High Resolution Timers system for the Linux kernel [14]. The high resolution timer system enables timers to interrupt between system timer ticks, which leads to a much more precise timer. The granularity possible with the high resolution system is only limited by the hardware clock used as source. Table 2.1 shows the results of running ping with 1000 packets between two hosts. Ref is the actual RTT of the test bed. The last four rows shows the round-trip time when NetEm is enabled and configured to have a 2 millisecond RTT. It clearly shows the need for having the High Resolution Timers system enabled in the kernel when running emulation experiments. Especially if the delay is low.

2.5 Dummynet

Dummynet is a network emulator originally developed for FreeBSD in 1997, designed for testing of network protocols [32]. Since then it has been greatly extended and it can now be used for a variety of applications [5]. With Dummynet it is possible to control the traffic going through network interfaces in many different ways. Configuring bandwidth and queue size limitations, using different queue management and scheduling policies and applying delay and losses are all available possibilities. This makes Dummynet an excellent tool for many different purposes, such as creating a bottleneck link for experimentation or using it as traffic control to manage a networks resources. Dummynet has been ported to other operating systems and are now available on, in addition to FreeBSD, Mac OS X, Linux/Openwrt and Windows.

2.5.1 Design

According to Rizzo [32], dummynet works on the principle that in order to simulate the presence of a network between two hosts some elements needs to be inserted in the flow of data. One element is routers with finite queue size and a queueing policy. Another is that the communication links needs to have a given bandwidth and delay. In addition to these two elements it is also important to be able to introduce packet reordering and losses as networks can have multiple paths and noisy links. Most of this is enabled in dummynet by the use of objects that is called a *pipe*. There are some emulation possibilities that does not exist in Dummynet, such as duplication and corruption [28].

Like the NetEm emulator the Dummynet system as a whole exists in two parts. The first part is the engine doing the actual emulation, aptly named *dummynet*. The second part is *ipfw*, which is used as a packet classifier by the Dummynet system. IPFW is a stateful firewall in FreeBSD that also provides traffic shaping, packet scheduling and in-kernel NAT [13]. In the Dummynet system it is both used as a packet classifier and as the main user interface to configure ipfw and dummynet.

Packets are passed to the classifier from various points in the network stack as can be seen in figure 2.10. This is usually done while processing a packet in layer two and/or three, both in the inbound and the outbound path. After a packet has been through a pipe it is injected back into the network stack right after the point where it was intercepted in the first place, unless it is re-injected into the classifier as explained in section 2.5.3.

2.5.2 Pipes - delay and rate control

Pipes are the basic object used in dummynet, it consists of a fixed size queue and a link with a given bandwidth and propagation delay.

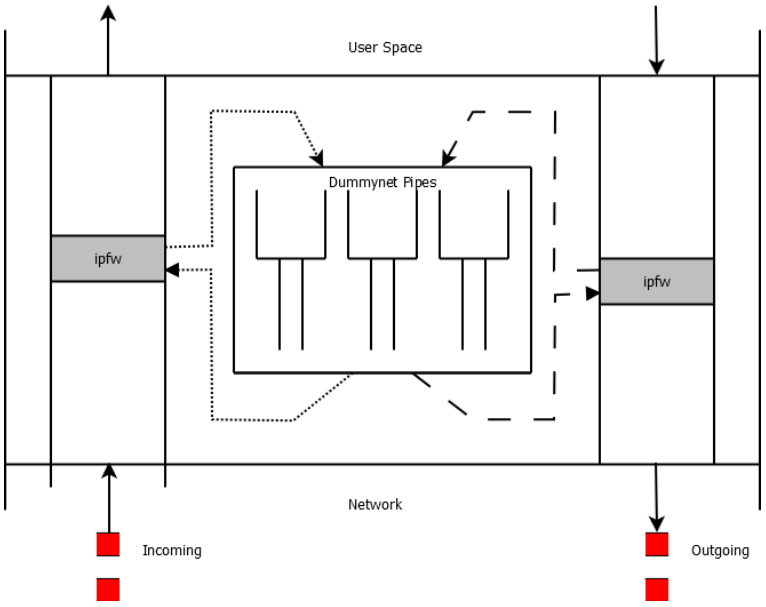


Figure 2.10: Packet intercepted by ipfw and sent through dummynet pipes

The number of pipes is only limited by the available memory and in combination with ipfw's packet classifier, explained in section 2.5.3, it is possible to create increasingly complex scenarios.

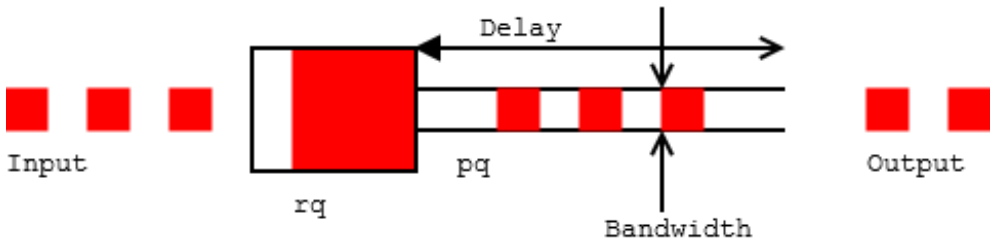


Figure 2.11: A dummynet pipe

Figure 2.11 shows the structure of a pipe in dummynet. The pipe consists of two queues, *rq* and *pq*, used to control the bandwidth and delay of the emulated link. A packet arriving at a dummynet pipe will first be queued in the *rq* queue as long as it is not full. The *rq* queue size can be configured and works as a routers finite sized queue. Packets are inserted into this queue according to a given queueing policy (The default queue in dummynet is a FIFO queue). Packets in *rq* are drained at a rate corresponding to the configured bandwidth B and placed in *pq*. Packets will remain in *pq* for a configured time t_D, which serves as the propagation delay of the emulated link, before they are injected back into the network stack. The moving of packets from *rq* to *pq* and from *pq* back into the network stack is achieved by running a periodic task moving packets conforming to the given configuration. All packets going through the pipe

will therefore be delayed by a time T defined by:

$$T = (l_i + Q_{rq})/B + t_D$$

Here l_i represents the length of the packet going through the pipe, Q_{rq} represents the occupation of the *rq* queue when the packet is enqueued, and B and t_D represents the bandwidth and delay of the emulated link. As an example, a 1000 byte packet going through a 1mbps link with a delay of 50ms when the queue is empty would give us: l_i = 1000, Q_{rq} = 0, B = 125000 Bps and t_D = 0.050s. This would result in a total delay time T of 58ms for a 1000byte packet to go through the pipe.

It is possible to configure and dynamically reconfigure pipes easily with one-line commands using ipfw. All pipes must be given a unique numeric identifier when created. Configuring the pipe in the example above with a queue size of 50 packets would be done with:

```
ipfw pipe 2 config bw 1Mbit/s delay 50ms queue 50
```

2.5.3 Packet classification

Dummynet needs a way to pass traffic from the network stack into the pipes to make use of them. This is handled by the packet classifier in *ipfw*. This classifier uses a set of numbered rules to match packets and decide where they should be passed to, the set of rules is called the *ruleset*. The overall structure of the command to insert a new rule to the ruleset is:

```
ipfw add rule-number action options
```

When a packet is passed to the classifier it will be evaluated against the rules in the ruleset in *rule-number* order until a match is found. This means that it will match packets against the lowest numbered rule before moving on to a higher numbered one, i.e. number 10 before number 20. All rules contain zero or more *options* that are used to match the packet. This can be a huge variation of criteria, ranging from such things as matching the address of a packet to matching against what interface the packet arrived on. If all the options of a rule matches the packet in question, it will be handled according to the *action* specified in the rule. Actions can be a variety of things, such as dropping the packet or passing it to a pipe. An example set-up with a combination of a pipe and a rule would be:

```
ipfw pipe 2 config bw 1Mbit/s delay 50ms queue 50
ipfw add 10 pipe 2 in proto tcp dst-ip 192.168.1.10
```

All packets sent to the classifier will be matched against this one rule. The options part of the rule is *in proto tcp dst-ip 192.168.1.10* while the action part is *pipe 2*. This rule has three options that all must match for the rule to take effect: *in* matches all inbound packets, *proto tcp* matches all packets

using the tcp protocol and *dst-ip 192.168.1.10* matches all packets that has a destination ip 192.168.1.10. Combined with the action means that this rule will send all incoming tcp packets going to 192.168.1.10 to pipe 2 where the packets will be subjected to a bandwidth limitation of 1mbps and a delay of 50ms before they are re-injected back into the network stack.

Packets not matching any rule in the ruleset will be placed back in the network stack as if they had not been intercepted at all. There is, of course, a possibility of creating rules that matches all packets, which can be used to handle all packets not matching any other rule.

Dummynet also supports the possibility of re-injecting packets back into the classifier when they are done in one pipe. This makes it possible to create increasingly complex topologies by simply looping the packets through a number of different pipes. Packets arriving in the classifier this way are subjected to the regular matching against rules and subsequent actions, such as putting the packet into another pipe.

2.5.4 Jitter and Reordering

In a real wide area network there is a high chance that there are multiple paths between two hosts. This can lead to variation in delays (jitter) and packet reordering. Jitter and reordering are linked in the way that jitter can often lead to reordering and vice versa. Think of a packet traversing a network with multiple paths between two hosts. The paths will most likely not be the same length, meaning that packets will use different time from one end to the other depending on which path they take. With the way a network routes packets there is a possibility that packets in the same stream will take different paths, for example, if one of the paths become highly congested. Packets from the same stream using different amounts of time will lead to both jitter and reordering. There is, of course, a possibility to have one without the other, such as a connection suddenly slowing down because of congestion when there are no other paths available before picking back up.

There is dedicated code to emulate these functions within NetEm, but just like with duplication and corruption, this is not the case with Dummynet [28]. However, while Dummynet does directly support emulation of jitter and reordering with dedicated code, there are different ways to achieve this. One option is to dynamically change the dummynet pipes as has been done by Armitage et al. [1]. By randomly varying the delay of the pipes traffic were flowing through at regular intervals they were able to achieve jitter. Some care must be taken with how the pipes are reconfigured as they experienced some troubles: *"Simply fluctuating the dummynet pipe according to a uniform random distribution can create unexpectedly non-uniform distributions of actual latency through your bridge*. Another option to emulate packet reordering and jitter is to use the

packet classifier, with an option to match packets with a given probability to randomly direct packets into different pipes.

As an example of achieving reordering and jitter consider the following rules:

```
ipfw add 100 prob 0.33 pipe 1 src-port 9001
ipfw add 200 prob 0.5  pipe 2 src-port 9001
ipfw add 300 pipe 3 src-port 9001
```

This set of rules will send all packets with a source port of 9001 to pipes 1, 2 or 3. the first rule will be checked before the second and so forth. Packets arriving with a source port of 9001 will have a one third chance of being sent to pipe 1 through rule 100. What must be kept in mind is that the the later rules will be checked after the first so rule 200 with the one in two chance of sending the packets to pipe 2 is of the remaining two thirds after the first rules has been checked. This means that the second rule also gives a one in three chance. The last rule, number 300, matches all packets with source port 9001 and will send all packets that is not matched with rule 100 and 200 to pipe 3. Combining this probabilistic match with several different pipes makes it possible to emulate reordering and jitter.

Yet another way of achieving reordering and/or jitter is possible by using the method of re-injecting packets that leaves a pipe back into the packet classifier. By sending some of the emerging packets through another pipe a measure of reordering will be achieved.

2.5.5 Packet loss

Loss of packets in a network usually happens because of overflow in the queues or some sort of queue management system (for example RED described in section 2.3.3). These types of packet loss cannot be specified in Dummynet, but they can easily be achieved by sending enough traffic through the emulator for this to happen naturally. A high amount of traffic means that the queues will fill up and losses will start to occur because of queue overflow. Queue management policies that cause loss even before the queue is full also exist in dummynet and will be explained in the next section.

Other possible causes for packet loss exist, such as radio link noise and interference. These can be emulated by the option to match packets with a given probability. The deny action of the packet classifier will drop all packets matching the criteria of the rule created. This will cause uniform random loss patterns in the emulated link. An example rule to add a 10% chance of dropping all tcp packets would be:

```
ipfw add 400 prob 0.1 deny proto tcp
```

2.5.6 Queue management and packet scheduling

The default queue in Dummynet is a simple FIFO queue with configurable size in number of packets or in bytes, but Dummynet also support the RED (Random early Detection) queue management algorithm and its gentle variant GRED. All of these are configurable to get the wanted behaviour from the queue.

In addition to changing the queue management algorithm there is also a possibility of doing packet scheduling with Dummynet. This is supported by creating a basic object called a *queue*. This queue object is used to create physical queues to store packets from different flows that can be used by the scheduling algorithm. To create a *queue* object to be used with packet scheduling the following command must be used:

```
ipfw queue N config sched X weight Y mask ...
```

This command creates a "queue" and also links it with the scheduler to queue is to be attached to, it also configures weight, or priority, for the scheduling algorithm. The optional mask parameter is used to group packets into different flows. The FIFO, WF^{2}Q+[1], DRR[2] and QFQ[3] scheduling algorithms are all supported by dummynet [5]. There also exists a simple API that enables easy creation of new schedulers if something else is needed. Figure 2.12 gives a graphical representation of the Dummynet system when configured with a queue, a scheduler and a pipe.

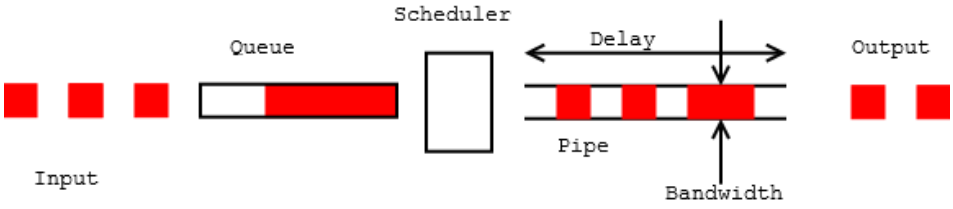


Figure 2.12: Dummynet system with a queue, a scheduler and a pipe

Configuring a system such as the one given in figure 2.12, where a regular FIFO queue is used together with QFQ scheduler and a pipe with 2mbps bandwidth could be done with the following commands:

[1]WF^{2}Q+: this is a scheduling algorithm where each flow is associated with a *weight* that specifies what share a flow will have of the capacity of an output link. WF^{2}Q+ schedules packets based on the weight and a virtual time function [2]

[2]Deficit round robin: a scheduling algorithm that that uses a modified weighted round robin approach. This algorithm serves all non-empty queues in a round robin fashion where each queue can dequeue a number of bytes up to a given *Quantum*. The remaining amount of bytes will be saved in a variable called *DeficitCounter* that can be used in addition to the Quantum in the next round of dequeuing [34]

[3]Quick Fair Queueing: is a very fast variant of WF^{2}Q+ [6]

```
ipfw pipe 2 config bw 2Mbit/s
ipfw sched 2 config type qfq
ipfw queue 20 config weight 10 sched 2
ipfw add 100 queue 20 out proto tcp
```

The first command creates the pipe, the second creates the scheduler and connects it with the pipe. The third command creates the queue and connects it with the scheduler. The last command tells the classifier to send all outgoing tcp packets through the system we just created.

Chapter 3

Design and Implementation

This chapter will first present the challenge of using TC and NetEm. This will be followed by a design proposal for our version of NetEm with a built-in rate control. Next we will give a thorough explanation about the implementation of the proposed design. At the end we will go through the resulting packet flow through the different NetEm versions.

When work on this thesis started the rate extension added to later versions of NetEm did not work, our work is therefore heavily based on the NetEm version included in Linux kernel 3.2. We will call this NetEm version *NetEm v3.2*, or simply *v3.2*. In Linux kernel 3.8 the rate control extension was working, hereafter referred to as *NetEm v3.8* or *v3.8*. We will therefore give a short explanation of this version so we can compare our implementation to an alternate way of implementing rate control in NetEM.

3.1 The TC/NetEm challenge

NetEm, as described in section 2.4, is developed to provide the possibility to emulate the properties of wide area networks, and has been a part of the Linux kernel since version 2.6.8. By providing these emulation capabilities it is possible to use NetEm to do research on networks, and test new protocols and applications in an artificially created environment that tries to match some real world network that is not easily tested in a repeatable and controlled way. Doing these kinds of experiments usually also needs some way to control the available bandwidth in the network, which is not something that was available in NetEm. A combination of two queueing disciplines can be used to achieve both network emulation (by using NetEM) and bandwidth management (by using some sort of rate limiting queueing discipline such as TBF), and is infact what resulted in this thesis.

3.1.1 Configuration

Some easy experiments sending traffic between two hosts, using a router to emulate a link with low bandwidth added by TBF and additional delay added by NetEm, revealed some wierd results when the bottleneck was congested. The general observation showed much higher delays than those added by NetEM. After digging through the Linux networking code, and the code of the queueing disciplines, the culprit was found to be the default queue size of NetEm. While in hindsight this was kind of obvious (the possibility of sending roughly 83 full packets per second on a 1mbit/s link, and the queue size of 1000 packets, will result in delays over 10 seconds when the bottleneck gets congested), it was not before a thorough understanding of the queueing disciplines and the underlying network code that the answer became glaringly obvious. While the issue in this experiment was kind of trivial, it still revealed a greater problem: using two queueing disciplines in conjunction to achieve network emulation makes for tricky configuration, and a high level of knowledge about the queueing disciplines and the surrounding network code is needed to make the system conform to the users expectations.

When configuring a system with two qdiscs, one must be the root qdisc while the other is a leaf qdisc. An overview of such a system can be found in figure 3.1. Consider the example where Qdisc 1 is the root qdisc and Qdisc 2 is the leaf qdisc. As explained in section 2.3.1, when queueing a packet into such a system it will always try to enqueue the packet at the root qdisc and likewise dequeue from the root qdisc. In this case the system will try to enqueue the packet at Qdisc 1. As long as the qdisc rules allow it to enqueue the packet it will try to insert it into its queue, which is now Qdisc 2. Qdisc 2 will then, if its rules permit, insert it into its queue. When dequeuing the same will happen all over again. The system will try to dequeue from Qdisc 1. If permitted by its rules it will try to dequeue a packet from Qdisc 2, which again will only release a packet if its rules permit.

Consider the following commands that create a root TBF qdisc where the default queue is changed to a NetEm qdisc used in the experiment mentioned above:

```
tc qdisc add dev eth0 root handle 1:0 tbf rate 1mbit \
                                burst 1540 limit 15400
tc qdisc add dev eth0 parent 1:1 handle 10: netem delay 100ms
```

The first line creates the root TBF qdisc. It is configured to have a rate of 1mbps, the limit which is the queue size is 15400 bytes, meaning that we have space for at least 10 full packets. The burst is set to 1540 bytes which means we can never have more tokens available than enough to instantly send one full packet. In itself this is an okay set-up, and might be exactly what is needed. A maximum rate of 1mbps and a queue with enough

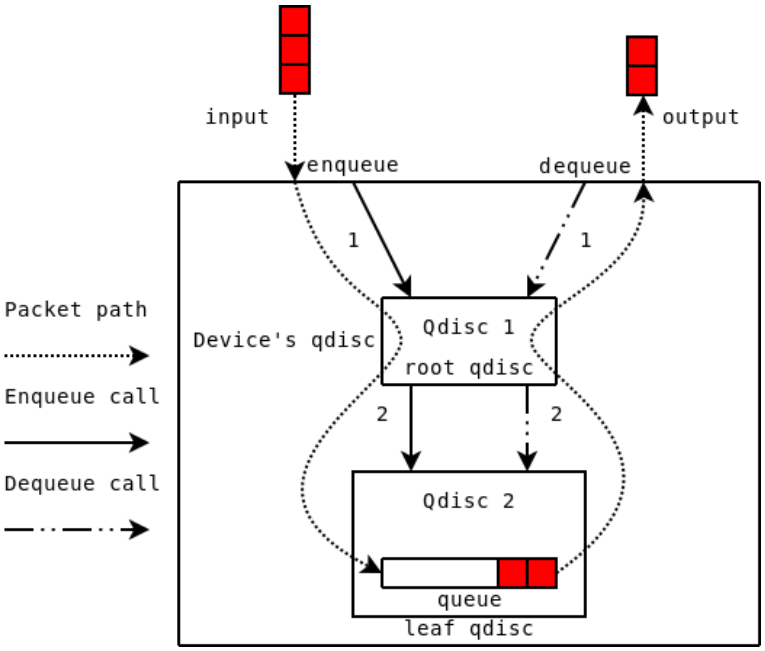


Figure 3.1: Overview of a system with two qdiscs

space for 10 full packets. The second line adds a NetEm qdisc to this system. It will simply delay all packets with 100 milliseconds. Again this seems okay. We should now have a system with a maximum rate of 1mbps a 10 packet sized queue and the additional delay of 100ms. This is not the case however. As can be seen from figure 3.1, packets are only queued in the leaf qdisc. As the NetEm qdisc is the leaf qdisc it will replace the queue of TBF. The limit placed on the TBF queue are no longer in effect as NetEm has its own queue which defaults to 1000 packets. So the system now consists of a maximum rate of 1mbps, a delay of all packets by 100ms and a queue size of 1000 packets. Congestion in this system will quickly lead to delays of over 10 seconds. From this simple example we can see that it is highly important to know what queues will be used to store packets and consequently how to configure the qdiscs. Using two queueing disciplines to configure one queue is also something that unnecessarily complicates the configuration process. Even if the two qdiscs are configured correctly the emulated system might not display the behaviour of a real system, which will be explained below. A system which enables the configuration of NetEm with rate control in only one qdisc seems like a great option to mitigate some of these configuration complexities and problems.

3.1.2 One queue mash-up

Network traffic travelling from one host to another through a network will first be subjected to the bandwidth of the link, before being affected by the transit delay. Consider a normal system with two machines placed quite

far apart. The direct link between them is 1mbps and has a one way delay of 100 milliseconds. When sending packets from one host to the other, the packets would first have to be sent out on the link. This is what, in an emulated system, would equal the rate control. When the packet is on the link it would take it 100ms to reach the other host, this is the transit delay where the packet is actually on the physical link. Once a packet leaves the sender, it will use 100ms to reach the receiver. This is time where the packet is not really buffered anywhere. The queue in this system would therefore not have to take into account the packets that are really out on the link travelling to its neighbour.

To emulate the same system, using two machines with a link of 1gbps and a delay of 0.1ms, we would first have to use some form of rate control to not send traffic at a speed higher than 1mbps, and we would have to hold all packets for 100ms to insure the correct delay. With the TC/NetEm system we only have one queue. This queue must therefore be able to both act as a real queue that holds packets ready to be sent out on the network, and hold the packets for the specified delay.

In a real network there are many type of queues, and their management schemes (such as FIFO or RED), which can have an impact on the traffic flow. What they all have in common is that they only work on buffered packets, not packets that are out on the physical link. In a system emulated with TC/NetEm, all packets must be stored in only one queue. This leads to the fact that the queue must not only handle the packets buffered for release to the network, but also the packets that in a real system actually are out on the physical link. As the packets being in transit should not actually be in the queue this presents possible behaviour differences between the real and the emulated system. Configuring the emulated system to behave like a real system is hard, if not impossible and some of the considerations that must be taken into account are explained below:

- **Queue management:** In a real system, queue management will only work on packets that are buffered in a queue to be sent out on the network. The packets being in transit, are actually on the physical link, and should not have an impact on the queue. In TC/NetEm, as all packets are in the same queue, this is not the case

- **Dropping:** Packet dropping in a real wired network are usually as a result of overflow in the queues, queue management systems (for example RED), or problems with the routing. To get the most lifelike behaviour this is something that should be handled by sending enough traffic to make overflow happen, or by using actual queue management on the emulated queues in the system. The emulation provided by NetEm should be therefore be used to emulate other causes for drops. This could be things such as radio link noise and interference. This leads to the fact that packets already being on the

physical link, the ones being delayed in the emulated system, should not have an impact on the packets that are waiting in the queue.

- **Reordering:** With NetEm, reordering can happen in two ways. It has a configuration option that will either reorder every Nth packet, or it can reorder based on a probability. This reordering is done by placing the packet at the front of the queue. Since the delayed packets are in the same queue, this will cause the packet to arrive much earlier than it should. The other way to have reordering with NetEm is to specify a jitter value along with the delay. Because NetEm employs a tfifo queue, all packets will be sorted by the time to send, and NetEm will therefore reorder packets when the delay value has a high amount of jitter. This reordering does not work however if another qdisc is used as the leaf qdisc. NetEm's own tfifo queue sorts packets based on their timestamp. When another qdisc is used as a queue, the timestamp will not be taken into account. In the case of TBF, all packets will be placed in order. This leads to packets further back in the queue having a lower time to send, but as it dequeues from the head of the queue they must wait for their turn. With a network where there is only one path from sender to receiver this might be wanted behaviour, but it still illustrates the need to have a high knowledge about how the qdiscs interact.

The test set-up in figure 3.2 uses two computers. One to send packets and one to receive them. TC/NetEm is used on the sender machine to emulate a slow link between these two hosts. When a program wants to send a packets it will do so through a socket. The networking stack tracks how much data is waiting to be transmitted through any given socket no matter if it is queued in the network device, in a queueing discipline or in the netfilter code. This means that all packets being sent through a socket will have that socket as its owner. The amount of memory in use by a socket is kept in the socket's *sk_wmem_alloc* variable. There are mechanisms in the Linux kernel that rely on this sk_wmem_alloc value, such as TCP small queue [24]. It will limit the amount of data that can be queued by any TCP socket. As packets being delayed by NetEm should not actually be counted as queued, as in a real system they would actually be out on the link. To be able to keep the packets in its queue, but not charge the sockets buffer, NetEm will call *skb_orphan()*. This function is used to remove a skb from its owner by calling the owners destructor function. The skb will still exist but it will be removed from its owner, meaning that it will not use ut space in the sockets send buffer. Usually this orphaning is done once the packet is transmitted by the network device. This means that it should happen after the rate limiting, but before the link transit delay. With one queue this is not possible as both rate limitation and delay is being done at the same time.

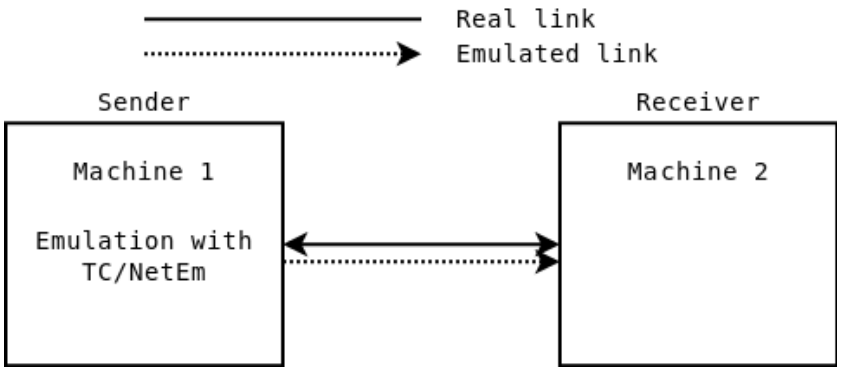


Figure 3.2: Two machine test bed

A system that avoids this problem is illustrated in figure 3.3. Here a machine is configured as a router that will just route packets between machine 1 and machine 3. Packets being routed will not be owned by any socket and therefore not use any socket buffer space. With NetEm being timer based, it works best if it can send packets as close to their time_-to_send as possible, it can be concluded that this set-up will be a better system to do research and testing with. If only to minimize the possibilities of interference from other tasks being run on the computer. While the problem of orphaning is removed by having a dedicated emulation machine it still do not stop it from being used on the same machine creating the traffic, so the problem should still be fixed.

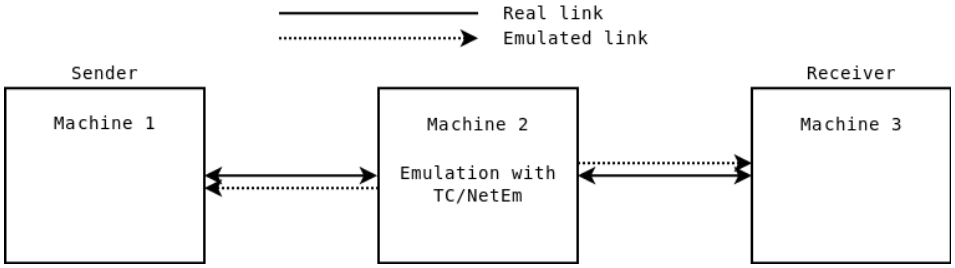


Figure 3.3: Three machine test bed

3.1.3 Traffic control is not network emulation

Another point to be taken into account when using TC is that it was not developed to be a network emulation tool. It was, as it name states, developed as a tool to do traffic control in the Linux Kernel. The traffic control system focuses on providing control over network services to provide some form of quality of service. This is important since the competition for network resources are usually very high. TC is used together with a number of different queueing disciplines to achieve this, and it provides a wealth of different options and configurations. NetEm

was developed as an enhancement to the traffic control features in Linux to provide network emulation for testing protocols and doing research. Despite its configuration problems documented above, it can work well in controlled situations as shown in [17, 28, 21].

A problem with NetEm though, is that it did not originally provide bandwidth emulation. To have rate control in an emulation set-up, an additional queueing discipline had to be used. The queueing disciplines that provide rate control are all built with the idea of restricting bandwidth based on the idea of traffic control. Take TBF as an example. It was originally developed to provide quality of service in a network, and uses a complex token bucket algorithm. It can at times allow for bursts of packets to go through at a higher rate according to the idea that there is no need to delay packets if the line is idle. This is, of course, not ideal in an emulation scenario. As emulation is used to mimic physical properties, the rate set in the emulator is supposed to imitate a physical link. This link will have a maximum rate that there is no physical possibility to exceed. This is something that can be mitigated by configuring TBF with its peakrate parameter. This is however something that adds more unneeded complexity to the configuration. In addition, the idea behind a token bucket implementation is that tokens fill up the bucket if there are no packets transmitted. This means that a packet arriving in a TBF queue when it has been idle for a time can be transmitted instantly. This is not something that should be possible in a real life scenario.

While the system proposed at the end of section 3.1.1 has been made available through the rate extension added to NetEm in the 3.3 version of the Linux Net-Next kernel, it is still using only one queue and is subjected to the problems described in the previous section. It does remove the problems described in this section, but it still paves the way for the design proposed in this thesis.

3.2 Design

In this section we will first present some design goals before we give a detailed description of our proposed design for a rate control extension to NetEm which we have named Double-queue NetEm.

3.2.1 Design goals

From the last section we can se that there are some problems when it comes to TC/NetEm. Configuring two queueing disciplines to work together is really complex and hard to do right. The rate extension added in version 3.3, and fixed in version 3.8 of the linux kernel, removes this problem as only one queueing discipline is needed to do network emulation. It still has the troubles linked with only having one queue doing all the

emulation. This is the basis for the two design goals below, which focuses on the creation of a built-in rate control to remove the use of multiple qdiscs:

- **Simple configuration:** To reduce the configuration complexity we want to have all emulation functionality in one queueing discipline.
- **Two queues:** With two queues we have the possibility to split rate control and delay emulation. This will mitigate the problems derived from the one-queue solution of NetEm 3.8.

3.2.2 Double-queue NetEm design

The design goals listed above lead to the design that will be explained in detail here, named *Double-queue NetEm*. It is a rework/extension to NetEm v3.2 which, which was explained in section 2.4. The actual implementation of the emulation features of NetEm v3.2 have been left mostly as is and will not be explained here.

The general idea behind the design is to provide the possibility to emulate channel bandwidth in NetEm without increasing its configuration complexity and at the same time remove the problems caused by doing all this with only one "physical" packet queue. This is done by splitting the queue used by NetEm into two queues, the *rate queue* and the *delay queue*. Inspiration for this design has been taken from Dummynet, described in section 2.5, which uses two queues and a periodic task that moves packet from one queue to the other.

When a packet is queued in NetEm it will first be placed in the rate queue. This will be done by calculating the time needed for a packet to be sent out on a physical link based on the packet's size and the configured bandwidth. When the packet has conformed to the bandwidth emulation it will be moved to the delay queue. Moving the packets from the rate queue to the delay queue will be handled by a timer function. A packet inserted into the delay queue will have to wait the calculated amount of time caused by the rest of the emulation features, before it will be released back to the network. An overview of the design can be seen in figure 3.4. The different parts of the system will be explained below:

- **Rate queue:** The basic function of the rate queue is to hold the packets until their *time_to_send* time has been reached. After this time the packet will be moved to the delay queue. This rate queue is designed to work like a the tfifo queue found in NetEm v3.2 and is bounded, which will force packets that arrive after the queue is full to be dropped.
- **Packet enqueue:** A queueing discipline provides an enqueue function. In this version of NetEm the main task of this function is to

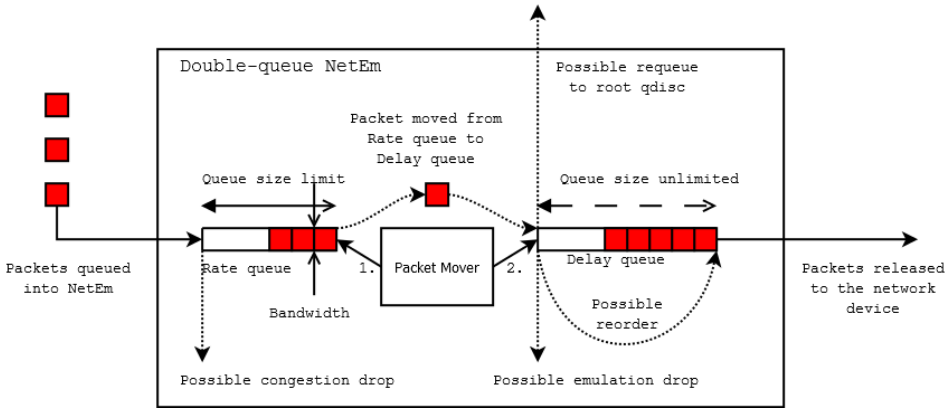


Figure 3.4: Double-queue NetEm design

place packets into the rate queue. It will calculate how much time a packet must wait based on the configured bandwidth and will then place it at the tail of the rate queue. The enqueue function must also check to see if the packet mover is running, if not it must start it.

- **Packet mover:** As the enqueue function of the queueing discipline will enqueue packets in the rate queue, and the dequeue function will remove packets from the delay queue, there must be a way to move packets from one queue to the other. This is done through a timer function, which will run only as long as there are packets in the rate queue. This function will move packets from the rate queue as long as their time to send is lower than the current time. When there are no more packets to move at current time it will do one of two things. If there are no more packets in the queue it will just stop, but if there are more packets to be moved at a later time it will reschedule itself to run again at the time to move the next packet.

- **Delay queue:** The basic function of the delay queue is quite similar to the rate queue. It must hold all packets until the time of their *time_to_send* value has been reached. A dequeue call to this queueing discipline will try to dequeue packets from this queue. This queue also works as the tfifo queue of NetEm v3.2, but it is unbounded. As it is unbounded, the delay queue can, in theory, grow to an infinite size. The amount of data that can be out on a physical link at any one time is, of course, based on the link's length and the channel bandwidth. As the delay queue is used as a representation of the connection between two hosts in a network, it should only be able to hold this amount of packets. As the amount of data in the delay queue will be bounded by the fact that the packet mover will only move as many packets as allowed by the configured bandwidth at any one time, in combination with the fact that packets are dequeued by the network stack once they have waited the emulated delay time.

- **Delay queue enqueue:** The delay queue is used to stage the packets before they are released back into the network stack. When the packet mover moves a packet from the rate queue it must enqueue the packet in the delay queue. Until now the only thing that has been emulated is bandwidth. The rest of the emulation is done here. The packet mover will call a function that calculates the configured emulation resulting in a new *time to send* value, the time that a packet can be released to the network. With this new value the packet will be enqueued in the tfifo delay queue in a time ordered fashion. It is worth noting that the time_to_send variable of a packet is changed when the packet is moved from the rate queue to the delay queue. There is no need to keep the old value once the packet has been moved.

- **Packet dequeue:** When the network stack wants to dequeue a packet from the queueing discipline it will call the qdisc's dequeue function. Double-queue NetEm dequeues packets much in the same way as NetEm v3.2 except as follows. When the network stack has dequeued a packet it will try to dequeue another one as long as the queue is not empty or the qdisc returns a NULL with the dequeue attempt. This means that a non-work-conserving qdisc (such as NetEm), which returns NULL when it wants to hold a packet longer, must restart the dequeue scheme at some time or there might be packets in the queue that will not be dequeued. If NetEm v3.2 can't send a packet, because it's configuration says the packet must wait longer, it will return NULL. This stops the dequeue process. NetEm v3.2 must therefore restart the process by using a qdisc watchdog timer, which it starts at the time to send value of the first packet in its queue. This version of NetEm will have two queues. This means that the dequeue function will have to additionally check the rate queue if the delay queue is empty.

3.2.3 Design discussion

There are a few points to be made from this design. The double queue approach might seem like a nightmare to configure, but this is not the case. In fact, it makes for an easier configuration. When configuring NetEm v3.2 with TBF we have to configure two qdisc, and know how they work together. When configuring v3.8 we have to take into account the queue length needed for both rate control and delay emulation. In Double-queue NetEm there is only one qdisc to configure, in addition when considering queue length there is no longer any need to think about other packets than the ones that are being rate controlled. We think this makes a compelling argument for our design.

In the two-queue design of Dummynet, the packet moving is handled

by a periodic function that just moves a variable amount of packets each time it is run, usually once every timer tick. This periodic moving of packets means that the overhead used to move packets stays the same. In Double-queue NetEm we have suggested a packet_mover function that calls itself with the time_to_send of the next packet to be moved. So each time the timer is triggered there might be only one packet moved. With the high granularity of the high resolution timers in the Linux kernel, this might potentially lead to an extreme amount of timer interrupts. A timer interrupt is not free, a certain overhead will be needed every time an interrupt happens, so this might pose a problem at higher packet rates.

Another point to think about is the separation of rate emulation and delay emulation. If modularized there is possibilities for easy implementation of changing the rate control method or introducing other delay, dropping or reordering schemes. One possibility would be a pattern matching dropping scheme.

3.3 Implementation

Much of the Double-queue NetEm version is based on NetEm v3.2. We will therefore first give an explanation of v3.2's implementation, before describing the changes made to implement our design. We have also used the packet length to tick time calculation of NetEm v3.8, so we will have a quick look at this version.

3.3.1 NetEm v3.2 implementation

A queueing discipline in Linux contains two main interfaces, one for queueing packets and one for releasing packets. In NetEm this is *netem_enqueue()* and *netem_dequeue()* respectively. There are also other interfaces which the system can use to interact with a queueing discipline, such as interfaces to create the qdisc (*netem_init()*), change its configuration (*netem_change()*) and so forth, but they are not important to understand how the qdisc works. The following describes how NetEm's features are implemented:

- **Configuration:** When created, all queueing disciplines will have a *Qdisc* struct created. This is where all the general information about a qdisc is stored, such as a pointer to the dequeue and enqueue functions, a pointer to the head of this qdisc's packet queue and other things that all qdiscs have in common. In addition, all queueing disciplines have their own private configurations. NetEm's private configuration and all its internal variables are stored in a struct named *netem_sched_data*. It contains a pointer back to the Qdisc struct, a qdisc_watchdog timer that can be used to restart the

dequeuing of packets and configuration variables such as latency and jitter.

- **Enqueue:** The process of queueing a packet into the NetEm qdisc is done through the *netem_enqueue()* function. When a packet is to be enqueued, NetEm will calculate a *time_to_send* value based on its emulation configuration. This time_to_send variable contains the time after which the packet can be released back into the network stack. If the packet has not been dropped it will be placed in the queue waiting to be released.

- **Default queue:** NetEm v3.2 actually contains the implementation of a basic queueing discipline called tfifo. This is a queueing discipline used exclusively by NetEm as its default queue. It is implemented as a FIFO queue with a modified enqueue function (*tfifo_enqueue()*). This function will take a packet and place it in the queue based on its timestamp. First it will check the timestamp of the packet at the tail of the queue as this is the most likely option. If this timestamp is lower if will use the following

```
skb_queue_reverse_walk(list, skb) {
  const struct netem_skb_cb *cb = netem_skb_cb(skb);
  if (tnext >= cb->time_to_send)
    break;
}
```

  This will find the place in queue where the packet should go, and results in a queue where packets are sorted from the lowest to the highest timestamp.

- **Loss and Duplication:** NetEm will first check for duplication with a call to *get_crandom()* which is a correlated random number generator. If there is to be duplication it will just increase a counter. To duplicate a packet, the skb needs to be cloned. As NetEm also have to check for drops there is no sense to duplicate a packet just to drop the original. So after duplication has been checked NetEm will see if there is to be a drop. This is done with a call to *loss_event()*. This will check to see if the packet is to be dropped by using the correlated random number generator or one of the implemented loss models. Loss will just free the skb and return while a duplication will lead to a cloned skb queued at the root qdisc. The packet must be enqueued at the root qdisc as the queuer only expects one packet to be enqueued.

- **Corruption:** Corruption is also check for with a call to the correlated random number generator. Corruption of a packet is achieved by:

```
skb->data[net_random() % skb_headlen(skb)] ^=    \
                                1<<(net_random() % 8);
```

which will just randomly introduce a bit error in the payload.

- **Variable delay:** To calculate the variable delay NetEm uses a call to the *tabledist()* function. This function will return a delay value based on the configuration. On one hand it could just return the constant delay specified, but in a complex scenario it could return a pseudo-randomly distributed value based on the constant delay and jitter configured. It uses a table lookup to approximate the configured distribution. The random elements in this function will use a uniformly-distributed pseudo-random source. The value returned from this function will just be added to the current time and stored in the skb's time_to_send variable.

- **Reordering:** NetEm v3.2 has two configurations for reordering, but both configurations does reordering the same way. By putting a packet at the front of the queue. The following code is used to check for reordering:

```
if (q->gap == 0 ||         /* not doing reordering */
  q->counter < q->gap ||  /* inside last reordering gap */
  q->reorder < get_crandom(&q->reorder_cor))
```

For the gap configuration, NetEm will just keep a counter so it can reorder every Nth packet. This will be checked for in the first two lines. Reordering by probability will just use the correlated random number generator.

- **Dequeue:** The process of dequeuing a packet from the NetEm qdisc is done through the function *netem_dequeue()*. To dequeue a packet NetEm will peek at the head of the qdisc used as its queue. This is done through *qdisc_peek()*, which is a virtual pointer to the peek function of NetEm's internal qdisc (by default this will be *qdisc_peek_head()* used by the tfifo qdisc). Then check the packets time_to_send variable against the current time. If time_to_send is less or equal to current time it can be released. This will be done by dequeuing the peeked packet and returning it to the caller of the *netem_dequeue()* function. If the time_to_send has not yet been reached NetEm will set up a qdisc watchdog timer that will start the dequeuing process at the time_to_send of the peeked packet.

```
qdisc_watchdog_schedule(&q->watchdog, cb->time_to_send);
```

- **Watchdog timer:** The watchdog timer is a part of the networking subsystem aimed at non-work-conserving queueing disciplines. When a qdisc cannot dequeue more packets for some reason (for example holding packets to emulate delay) it must stop the dequeuing process or it would waste resources trying to dequeue over and over again. To restart it the qdisc watchdog timers can be used. This system will restart the dequeuing process in the networking stack which will eventually lead to a new dequeue call (*netem_dequeue()*).

3.3.2 Double-queue NetEm implementation

This section will be used to show the implementation of the changes and additions done to NetEm v3.2 to implement the Double-queue NetEm design proposed above. The design featured a packet_mover function that would be started with a timer, and it would run separately from the enqueue and dequeue calls, moving packets from the rate queue to the delay queue. This raises the question about race conditions. The Linux networking system works in the way with qdiscs that a lock is taken on the qdisc's queue when either enqueue or dequeue is called. This made it hard to feature a locking scheme that would not deadlock when the packet_mover function is introduced as an independent part of the system. Different locking schemes where tried with varying success, but no scheme worked as needed. The main idea about the packet mover running when packets needed to be transferred was to have the time_to_-send be as realistic as possible as it was calculated when moving the packet at the correct time. With problems getting it to work correctly, a redesign on how the packets were moved from one queue to the other commenced. In the end we landed on a relatively simple set-up: Whenever NetEm's enqueue or dequeue functions is called we will start with a call to the packet_mover function. The only changes to the function is that it is no longer a callback function designed to call itself as long as there are packets in the rate queue. Now it will just move all packets that have an expired time_to_send value from the rate queue to the delay queue.

There are four main changes/additions to NetEm v3.2 that will be explored. Most of the *netem_enqueue()* functionality has been moved to the *delay_queue_enqueue()* as all of the old emulation will now be done when moving the packet from the rate queue to the delay queue. netem_-enqueue is used to queue packets in the rate queue. A *packet_mover()* is used to move packets from the rate queue to the delay queue, and the *netem_dequeue()* function works mostly as it did in v3.2, but it has to take into account the extra queue in the qdisc.

netem_enqueue()

The *netem_enqueue()* is now a steppingstone for the packets on their way to the rate queue. The first thing this function will do is to call *packet_mover()*. We call this function before queueing a packet to make sure that there is the correct amount of packets in the rate queue in case of congestion. To queue a packet in the rate queue, we need to calculate how long it must wait there based on the size of the packet and the configured rate. The method to calculate how much time a given packet uses to be transmitted is the same as the one used in NetEm v3.8, which will be explained in section 3.3.3. After the packet has the correct time_to_send it will be placed in the rate queue. It is important to note that the surrounding network code expects to find the amount of packets queued in the queueing discipline by looking at the queue length of its queue. This version uses two queues, and as the system only checks one it will give the wrong result. We have fixed this by updating the delay queue length (this is the one the surrounding system checks) when we enqueue a packet in the rate queue. When we move packets from the rate queue to the delay queue we make sure that we do not update the length of the queue again. We have implemented a dedicated code path in this function that works when Double-queue NetEm is used with no configured delay, it will just skip the rate queue and queue the packet directly into the delay queue.

packet_mover()

The *packet_mover()* function is called from both *netem_enqueue()* and *netem_dequeue()*. It is implemented to move packets from the rate queue to the delay queue. It uses *delay_queue_enqueue()* to do this. As we can see from the code below, it will continue to move packets as long as their time_to_send as lower than the current time.

```
while ((skb != NULL) && (netem_skb_cb(skb))->time_to_send <=   \
                                        psched_get_time()) {
        skb = __skb_dequeue(&q->rate_q);
        delay_queue_enqueue(skb, q->this_qdisc);
        skb = skb_peek(&q->rate_q);
}
```

delay_queue_enqueue()

Most of the functionality of *netem_enqueue()* in NetEm v3.2 is moved to this function in Dq NetEm. It works mostly in the same way. The exceptions are that it will not check for any queue size limit, and that it will not update the queue length stat of the delay queue if rate emulation is used.

netem_dequeue()

This function is implemented almost as in v3.2. The first difference is that it will call the *packet_mover()* function when it starts. The other difference is when there are noe packets that it can release. When NetEm cannot release a packet it will reschedule the dequeuing process by setting up a watchdog timer. If the delay queue is empty it must also check the rate queue in case there are packets there.

3.3.3 NetEm v3.8 implementation

There is really only one real change from v3.2 to v3.8 of NetEm, some small changes that have no impact on this thesis will be left out. The code to NetEm v3.8 can be found at [38]. V3.8 have the added rate extension. This is mostly implemented by calculating how much time a packet of a given size needs to propagate out on the physical link. This is mainly done in the *packet_len_2_sched_time()* function. This is also the calculation that is used in Double-queue NetEm.

```
static psched_time_t packet_len_2_sched_time(unsigned int len,
                                    struct netem_sched_data *q) {
        u64 ticks;
        len += q->packet_overhead;
        ticks = (u64)len * NSEC_PER_SEC;
        do_div(ticks, q->rate);
        return PSCHED_NS2TICKS(ticks);
}
```

In addition to this, as NetEm v3.8 uses one queue for both rate control and delay, it must calculate the time_to_send based on the last packet in the queue and the delay to be added. This is what was wrong in the versions with the rate control that did not work correctly.

3.4 Packet flow in NetEm

Here we will describe the flow through the different versions of NetEm.

3.4.1 NetEm v3.2 flow

The flow through the NetEm v3.2 queueing discipline is quite straightforward and an overview can be found in figure 3.5. As explained in section 2.3.1, a queueing discipline is first involved when the kernel tries to enqueue a packet in the qdisc through its virtual enqueue method.

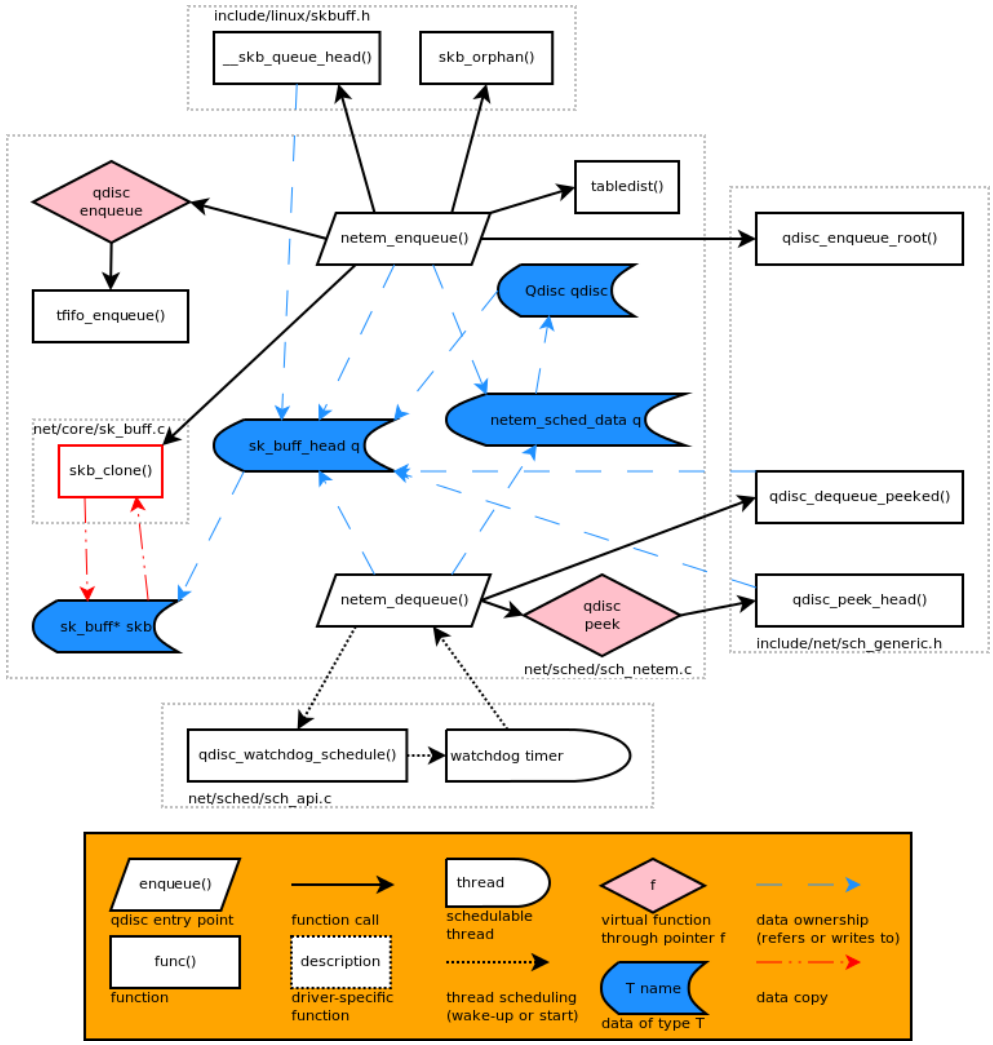


Figure 3.5: NetEm v3.2 qdisc flow

- **Enqueue:** In the case of NetEm, *netem_enqueue()* is called, which takes one packet and enqueues it into the qdisc. Through calls to various functions it will first orphan the skb (*skb_orphan()*), then do drop and corruption. Drop will just free the skb and return. Duplication will be achieved by the *skb_clone()* function to duplicate the packet and a call to *qdisc_enqueue_root()* to enqueue the duplicated packet at the root qdisc. What it does next depends on reordering. If a packet is to be reordered it will first set its time_-to_send to current time, then put it at the head of the queue with *__skb_queue_head()*. In the case of no reordering, it will calculate the packet's time_to_send based on the configured delay, jitter, correlation and distribution with a call to the *tabledist()* function. The packet is then placed in the queue with *qdisc_enqueue()*, which is a virtual pointer to the enqueue function of the qdisc used as NetEm's queue (by default it will be *tfifo_enqueue()*).

- **Dequeue:** When *netem_enqueue()* returns, the kernel will immediately call the device output queue through the use of *qdisc_run()*, which will in turn call *qdisc_restart()*. This function will call the qdisc's virtual dequeue function to dequeue a packet. In the case of NetEm this is *netem_dequeue()*. *netem_dequeue()* will peek at the first packet in its queue. If the packet is ready to be sent, its time_-to_send value is lower than current time, it will be dequeued from the queue with *qdisc_dequeue_peeked()* and returned to the caller of *netem_dequeue()*. If NetEm cannot release any packets, the peeked packet has a time_to_send higher than the current time, it has to reschedule the dequeuing. In this case NetEm will make use of the qdisc_watchdog system by calling *qdisc_watchdog_schedule()* with the time_to_send of the first packet in the queue as a parameter. *qdisc_watchdog_schedule()* starts a *watchdog timer*, which will restart the dequeuing process at the time of the first packet to be dequeued. This will eventually lead back to the *netem_dequeue()* function, and the packet that could not be sent earlier.

3.4.2 Double-queue NetEm flow

The resulting flow in Double-queue NetEm is a bit more complex than in NetEm v3.2. An overview can be found in figure 3.6. The functions to interact with the queueing discipline is still the same, *netem_enqueue()* to queue packets, and *netem_dequeue()* to dequeue them.

- **Enqueue:** *netem_enqueue()* is called when the network subsystem wants to enqueue a packet to the NetEm qdisc. First it will call *packet_mover()* to move all packets with an expired time_to_send from the rate queue to the delay queue. The packet moving flow

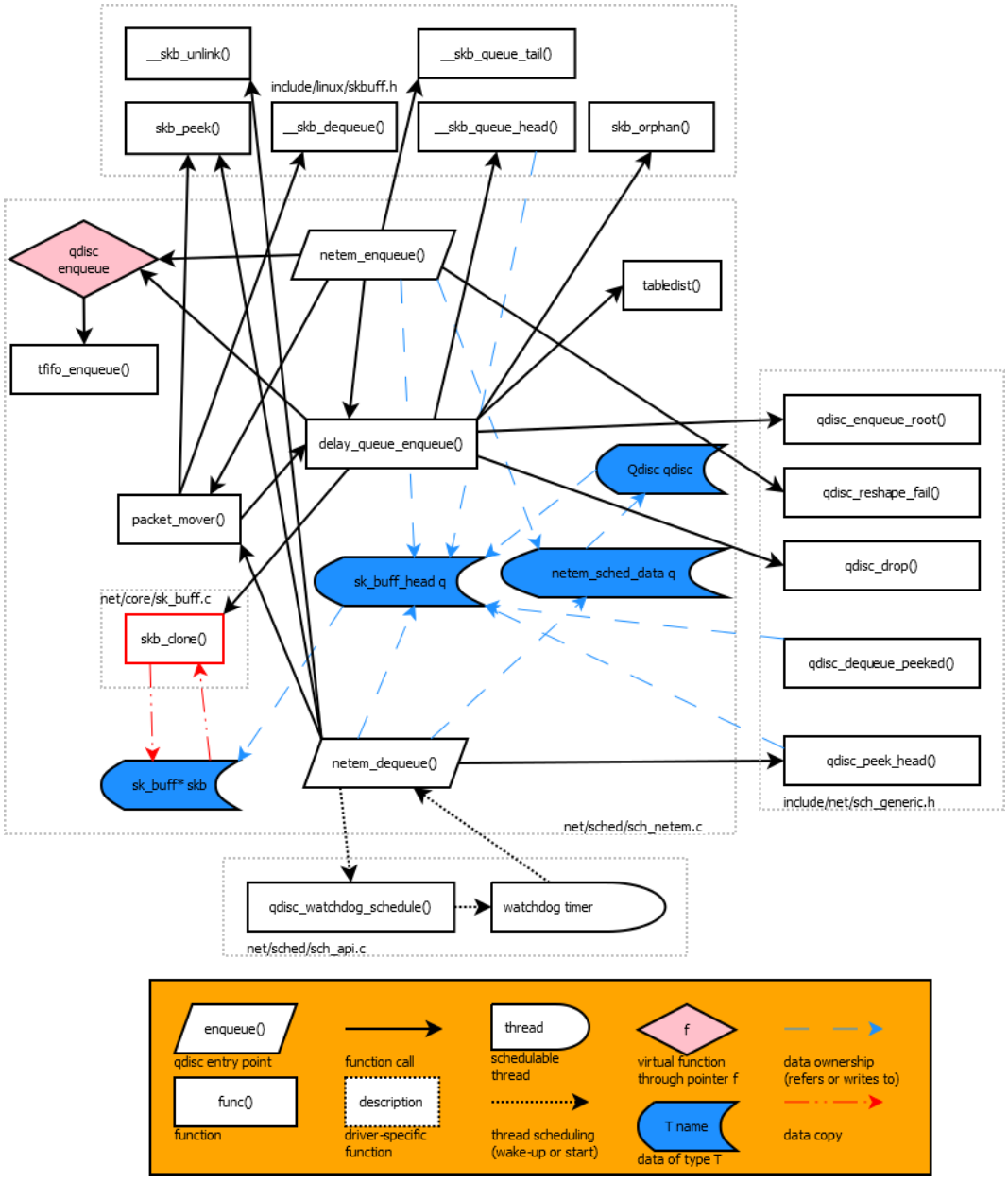


Figure 3.6: Double-queued NetEm flow

is described below. Next the packet to be enqueued will be added to the tail of the rate queue with *__skb_queue_tail()*. The *time_to_send* of the packet will be calculated first, so the queueing discipline knows when to move the packet from the rate queue to the delay queue. A dedicated code path exists in this function that will be used if Double-queue NetEm is configured without rate emulation. This code path will just queue the packet directly to the delay queue with a call to the *delay_queue_enqueue()*. The *delay_queue_enqueue()* function works much like the *netem_enqueue()* of NetEm v3.2. This means that it will do much of the same calls as explained in the last section.

- **Packet moving:** The *packet_mover()* function will move packets from the rate queue to the delay queue with *delay_queue_enqueue()*. This will continue to move packets as long as there is one in the rate queue with an expired time_to_send value. After which it will return to the callee.

- **Dequeue:** In *netem_dequeue()* there are also some changes to the packet flow. First it will call *packet_mover()* to move all packets with an expired time_to_send from the rate queue to the delay queue. This function works as described above. Next it will check the delay queue for a packet with *qdisc_peek_head()* and return the packet if it is ready to be sent. If there is packets in the queue, but the time_to_send is not reached yet, it will reschedule the dequeuing at the time of the first packet with *qdisc_watchdog_schedule()*. If there are noe packets in the delay queue it will check the rate queue with *skb_peek()*. If there is no packet it just returns as there are noe more packets in the qdisc, but if there is one it will reschedule the dequeuing with *qdisc_watchdog_schedule()*.

3.4.3 NetEm v3.8 flow

NetEm v3.8 does not have a very different flow path than v3.2. The only real difference is the rate calculation that will be added to the time_to_send of a packet by calling *packet_len_2_sched_time()*.

Chapter 4

Results

We wanted to see if our implementation works as we envisioned. We have therefore run several extensive tests and compared it to other versions of rate emulation. As the design and implementation have been done with the goal in mind to create a built-in version of rate control we will focus most of our tests on just that, the rate. In addition we have tested the delay parts of the emulator to see if the rate emulation addition would break the old functionality. We expect our implementation to at least mitigate some of the burstiness that might occur while using TBF to emulate bandwidth. The splitting of rate and delay emulation should also have an impact on our results.

4.1 Testbed

All tests in this thesis were conducted using the same testbed (shown in figure3.3), using three computers. Two of them were identical (Core 2 Duo, 3.0 Ghz, 4 GB memory). One was used as the receiver, while the other was used as the router. The third computer was almost identical (Core 2 Duo, 2.4 Ghz, 4 GB memory). It was used as the sender. The router did emulation on the outgoing interfaces to the sender and the receiver. All network devices was theoretically capable of 1 Gbps. Both the sender and the receiver used Linux kernel version 3.5.0, while the kernel used by the router changed between v3.2 and v3.8 depending on the NetEm version being tested.

4.2 Testing of Double-queue NetEm

The kernels used in the testing was as follows. NetEm v3.2 used the 3.2 kernel, while NetEm v3.8 used the 3.8 kernel. Double-queue NetEm used a 3.8 kernel with our implementation of Double-queue NetEm. The tests the were run with the interface doing the rate control was done using NetEm

v3.2 on the 3.2 kernel.

4.2.1 Rate control

The test

To test the rate emulation feature of Double-queue NetEm we have used UDP with a set of different test scenarios to see how the implementation works. Each test in the set had a configured bandwidth (between 1.5 Mbps and 100 Mbps) and a configured delay (between 0 and 50ms). We chose the test metrics as follows:

- UDP is a transport protocol that will just send data with no guarantees for delivery, ordering or protection against duplication. Lost packets will remain lost, as there is no resending of packets. In a perfect network, where none of these problems is an issue, UDP will work very well. No reactionary components, such as congestion avoidance in TCP, means that UDP uses very little overhead. With this in mind we expected to see very little impact from the protocol on our results, meaning that we should be able to emulate rates that are very close to the theoretical maximum.

- The test was done with three different bandwidths. First we had ADSL Lite which has a speed of 1.5Mbps/0.5Mbps. This is a common internet access technology recommended by the International Telecommunication Union (ITU) [19]. Sender to receiver was configured to 1.5Mbps while receiver to sender was configured to 0.5Mbps. The configuration of the receiver to sender emulation was really not necessary for this test, but with the way our test system is set up we needed to configure both ways. This does not impact this test, as UDP only sends traffic in one direction. We also chose to test with bandwidths of 10Mbps/10Mbps and 100Mbps/100Mbps as common Ethernet technologies provides network cards with these speeds. With these bandwidths we also had the ability to configure the device driver to run at these rates to provide a baseline to check the emulation results against.

- In this test we chose to configure different delays to make sure the stability of our emulation system does not change with different amounts of packets staged in the delay line. We used 0ms, 10ms and 50ms as end-to-end delays for these tests.

- To make sure the queue sizes of the different scenarios doesn't impact the results, we chose to set queue sizes based on bandwidth-delay product (BDP). BDP refers to the product of the data link's capacity and either its end-to-end delay or its round trip time (RTT).

It is calculated with bits per second for the capacity and seconds for the delay. Our queue sizes were set to BDP calculated with RTT to make sure the queues were large enough for the emulators to be able to send at the rate it supposed to emulate.

To generate the UDP traffic we used the program *Iperf* [27], to send data from the sender to the receiver. Iperf sends udp data in packets of 1470 bytes, which translates to an ip packet with the size of 1498 bytes when we include the ip (20 bytes) and udp (8 bytes) headers. In Iperf it is possible to configure a target bandwidth rate to send packets at. We set this to about 20% more than the configured test scenario to make sure that we had enough packets running through the system so that packet creation would not have an impact on the rates. In addition, this fills up the queues so we can see that there is no trouble with the queue implementation.

Below is the equation we have used to calculate our queue size Q_s, using bandwidth in bits per second B and delay in seconds D with regards to our packet size P in bits. The answer was rounded up to the nearest full packet:

$$Q_s = (B_{bps} x D_s)/P_b$$

Using this formula we could see that we needed a queue size of 17 packets when emulating a link of 10Mbps with an rtt of 20ms.

$$Q_s = (10000000x0,02)/11984 \approx 17$$

This of course left us with a problem to calculate the queues when there is no added delay. We say added, as in reality, when we are working with real systems that have real network links, there will always be some delay inherently in the network. The rtt of the reference system was shown in table 2.1. Again to get results based on the rate implementation without queue interference, we chose to calculate the scenarios with no added delay as if it had an rtt of 1ms. This left us with the test scenarios shown in table 4.1. All of these test were run with Double-queue NetEm and with NetEm v3.2 to compare against. In addition we ran the last 6 scenarios with setting the mode of the interface to compare against what a device limited rate would be.

Result

The statistics from the different rate test scenario streams are summarized in table 4.2. From the table we can see that the rates from NetEm v3.2 and our Double-queue (Dq) implementation are very close. At higher rates we can see that there is a bit more throughput with Dq than there is with v3.2. One possible reason for this might lie with how we have configured TBF in our experiments. TBF have many possible configurations, but as we do not want any burst over the targeted bandwidth, we have configured it

Rate (Mbps)	Delay (ms)	Queue size (packets)
1.5/0.5	0	2
	10	3
	50	13
10/10	0	2
	10	17
	50	84
100/100	0	17
	10	167
	50	834

Table 4.1: Rate test scenarios

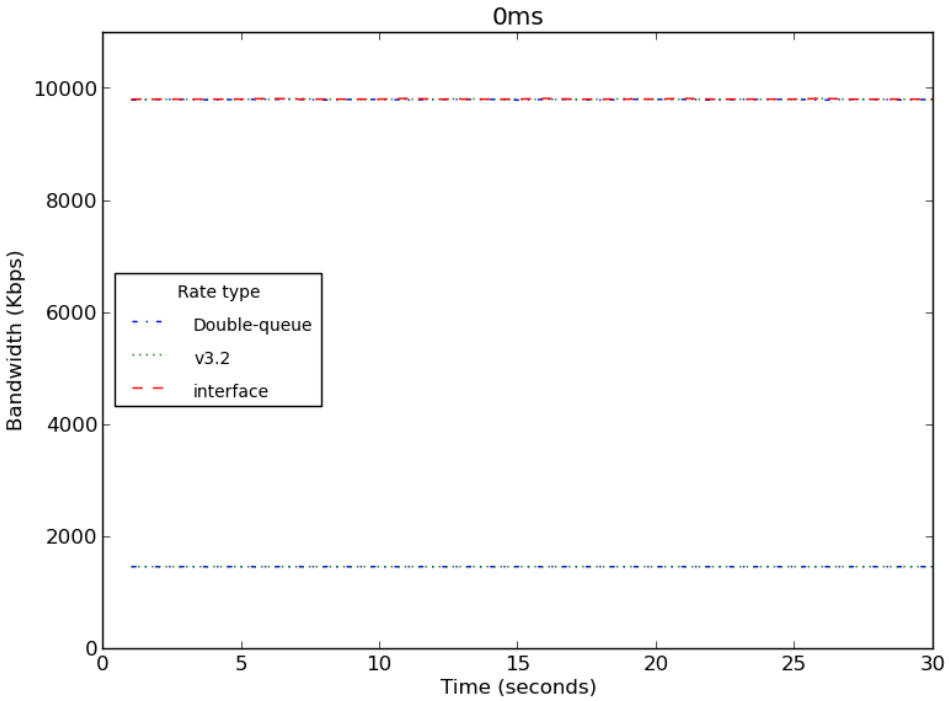
with a burst size of 1540 bytes. As we are sending only full packets, TBF should not at any point in time be able to burst more than one packet. With TBF never being able to send more than one packet it should also even out the release of packets eliminating much of the inherent burstiness of the implementation.

At 10Mbps and 100Mbps we have the possibility to compare the results of the two different rate implementations to the rate given when the interface is set with the corresponding bandwidth setting. Here we can see that we consistently get a higher rate with v3.2 and Dq NetEm. The reason for this has been narrowed down to the calculation of packet sizes. Our implementation is calculating the time a packet must remain in the rate queue based on the length of the packet which can be found in the packet's sk_buff struct. The *len* member of this struct keeps the size of the packet. This does not take into account the headers of the packet. As we send the headers with the packet across the network, they will of course also use bandwidth. In our set-up which uses UDP over IPv4 this equals 8 bytes for the UDP header and 20 bytes for the IPv4 header that is not included in the calculation leading to higher rates than configured. There exists a patch to a later version of the kernel that aims to correct this. It uses the pkt_len member of the qdisc_skb_cb struct (a control buffer used while the packet is inside a qdisc), to store the length of the packet with the headers included. We have not tested it, but as there is work that aims to universally fix the problem we have chosen to not correct for packet headers in our implementation. The last column in the figure contains the resulting throughput once we have adjusted for the headers. This is done by reducing every packet with 28 bytes to get the real throughput of the scenario. All the figures in the results have likewise been adjusted.

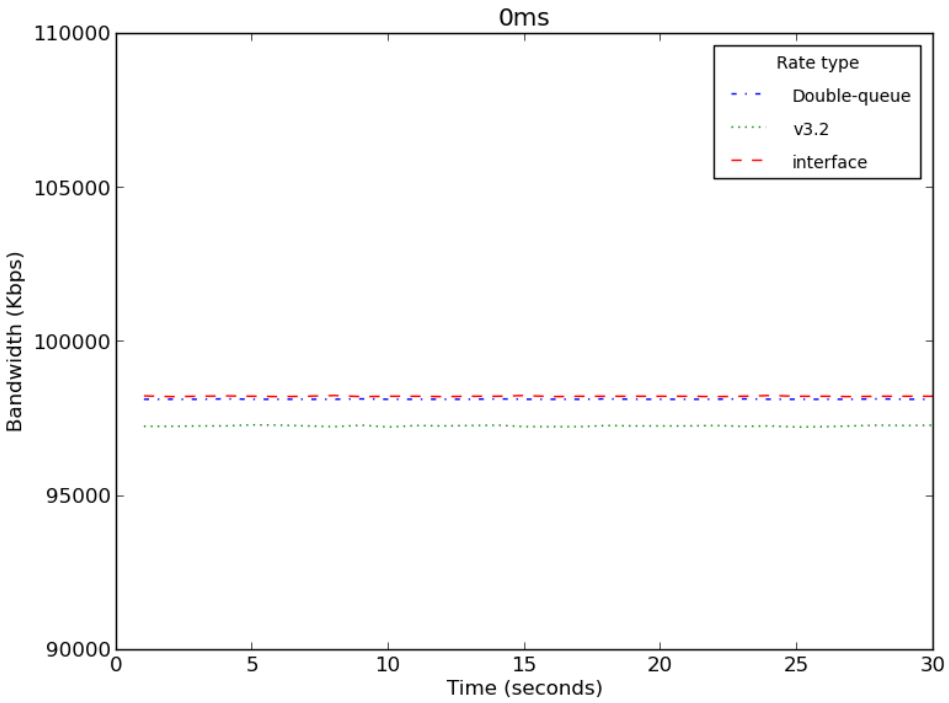
Figure 4.1a shows the throughput for the various rate control mechan-

Scenario			Runtime (S)	Total Packets	Total Bytes	Throughput (Bps)	Throughput (Mbps)	Throughput (Hdr Corr)
Rate	Delay	Type						
1.5Mbps	0ms	v3.2	31.0314	3849	5688822	183325	1.4666	1.4388
		Dq	31.0315	3848	5687344	183277	1.4662	1.4384
	10ms	v3.2	31.0316	3848	5687344	183276	1.4662	1.4384
		Dq	31.0211	3847	5685866	183290	1.4663	1.4385
	50ms	v3.2	31.0719	3854	5696212	183324	1.4666	1.4388
		Dq	31.2625	3860	5705080	182489	1.4599	1.4323
10Mbps	0ms	v3.2	31.0122	25647	37906266	1222302	9.7784	9.5932
		Dq	31.2531	25633	37885574	1212218	9.6977	9.514
		interface	31.1945	25334	37443652	1200329	9.6026	
	10ms	v3.2	31.0008	25637	37891486	1222273	9.7782	9.593
		Dq	31.2616	25648	37907744	1212598	9.7008	9.517
		interface	31.1931	25333	37442174	1200337	9.6027	
	50ms	v3.2	31.0628	25688	37966864	1222263	9.7781	9.5929
		Dq	31.0128	25632	37884096	1221562	9.7725	9.5874
		interface	31.2528	25377	37507206	1200124	9.6010	
100Mbps	0ms	v3.2	31.0098	254039	375469642	12108096	96.8648	95.0297
		Dq	31.0053	256307	378821746	12217969	97.7438	95.892
		interface	31.0263	251959	372395402	12002554	96.0204	
	10ms	v3.2	31.0185	254108	375571624	12107989	96.8639	95.0289
		Dq	31.0293	256457	379043446	12215659	97.7253	95.8739
		interface	31.0275	251944	372373232	12001406	96.0112	
	50ms	v3.2	31.2039	254044	375477032	12033014	96.2641	94.4405
		Dq	31.1071	257124	380029272	12216784	97.7343	95.8829
		interface	30.9815	251554	371796812	12000601	96.0048	

Table 4.2: Timer resolution results

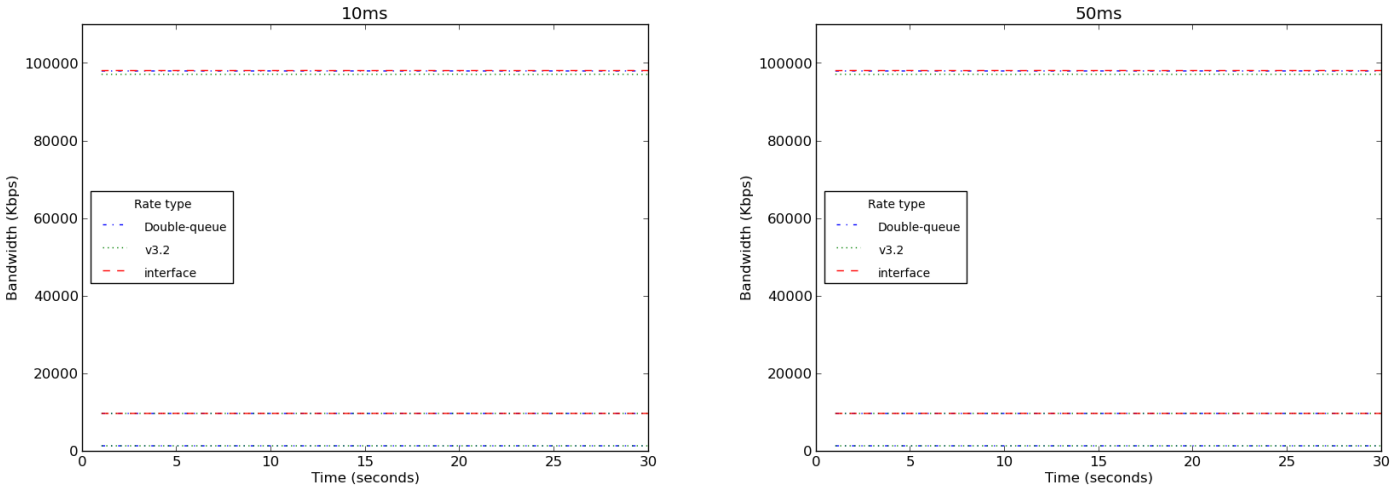


(a) 1.5Mbps / 10Mbps



(b) 100Mbps

Figure 4.1: Rate emulation with 0ms delay



(a) 1.5Mbps / 10Mbps

(b) 100Mbps

Figure 4.2: Rate emulation with 10ms and 50ms delay

isms for the 1.5Mbps and 10Mbps links with 0 delay configured. We can see that both v3.2 and Dq NetEm is very close at 1.5Mbps, and they are also both very similar to the interface at 10Mbps. Figure 4.1b the throughput at 100Mbps and 0 delay. Here we can see that Dq NetEm is very close to what the interface gives as rate. This is a very good indication that our implementation is working as it should. The v3.2 version of NetEm is a bit lower in this graph. The only reason we can find is the one we discussed earlier. The configuration of TBF might be the culprit. We could probably raise the throughput of NetEm v3.2 by allowing some bursts in TBF. This is however not something we want when running emulation, so we continued with the same settings. However, this illuminates the fact that configuring the queueing disciplines together as they are now, is a hard task. Figures 4.2a and 4.2b shows the same results when we use end-to-end delays of 10ms and 50ms respectively.

In figure 4.3 we can see a zoomed-in view of the scenario of a 100Mbps link with the end-to-end delay of 50ms. Here we notice that the v3.2 scenario, which uses the TBF qdisc for rate control, has a much more bursty behaviour than Dq NetEm. There seems to be some very small bursts in the stream with Dq Netem, but it is generally much better behaved than v3.2. We can see that Dq NetEm is very similar to the interface scenario, both in burstiness and in actual throughput.

The cumulative distribution function (CDF) in figure 4.4 shows the scenario with 100Mbps rate and 10ms delay. As we can see from the graph, all of the rate control mechanisms give a rate close to what is configured. This proved to be the case in all the tested scenarios.

The conclusion is that the Dq implementation proposed in this thesis performs very well in a rate setting. It reduces the inherent burstiness of

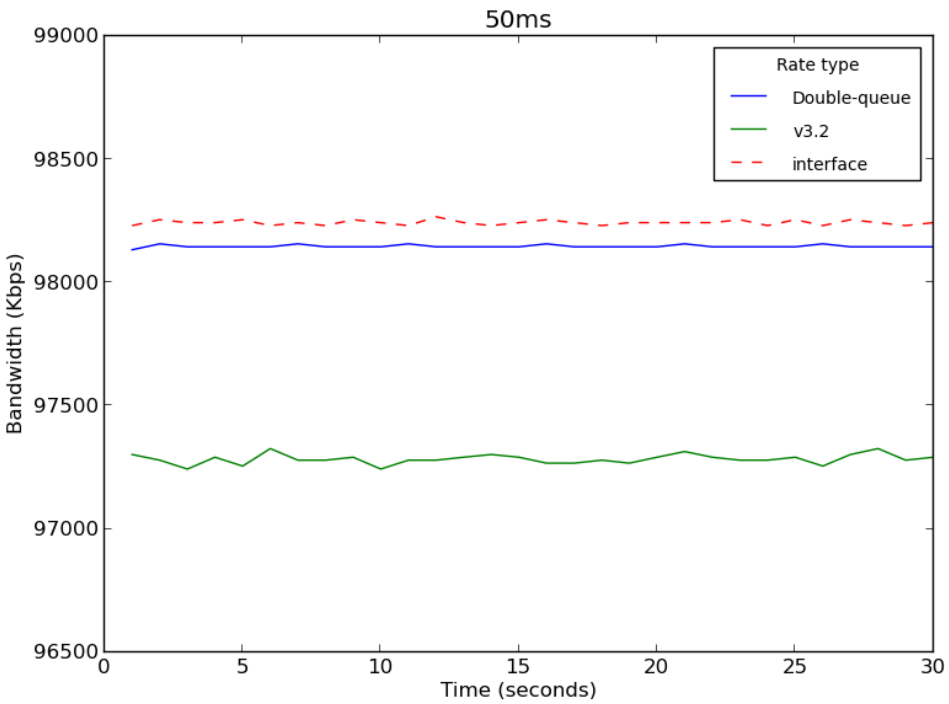


Figure 4.3: Rate emulation 100Mbps 50ms delay

the TBF rate implementation to make it more emulation friendly. Burst is as explained earlier not an ideal situation for a rate emulation scenario. When the correct packet size is computed, it also gives rates very close to what a real life link at a certain speed would give, as indicated by the comparison with the interface scenarios.

Double-queue NetEm vs NetEm v3.8

The main focus of our testing was to compare the Dq implementation against NetEm v3.2. We did decide to do a simple test to compare our version with NetEm v3.8. This is the version where the built-in rate control extension was fixed. We ran the test set again with same metrics that was used on the other versions. In figure 4.5 we can see the result of the test case with a 100Mbps link and a delay of 50ms. From the figure we can see that two versions behave almost exactly the same. The amount of burst and the throughput is almost identical. This is likely caused by the fact that both versions use the same time dependent approach to rate emulation. All tests in the set showed the same result.

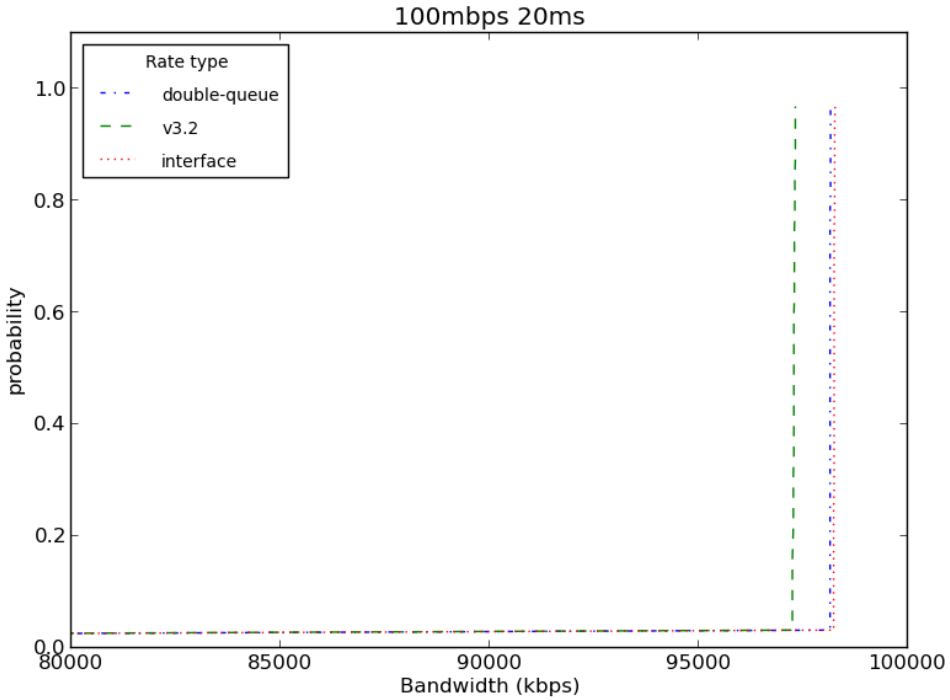


Figure 4.4: CDF of rate emulation with 100Mbps 10ms delay

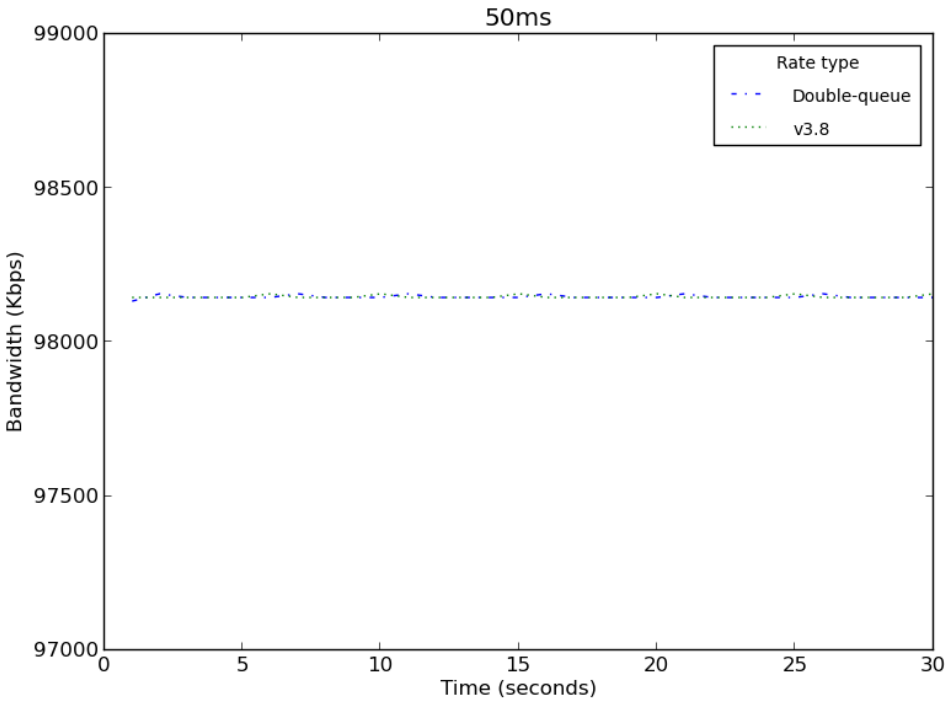


Figure 4.5: Latency vs. RTT

4.2.2 Latency

The test

Latency is an important part of emulation. To make sure that our addition of rate emulation to NetEm did not break this functionality we did latency tests with Dq NetEm comparing it to NetEm v3.2. The two test sets were both run with the rate emulated to 10Mbps. One test set was done with the *Ping* tool provided by the *inetutils* package [15]. Ping send an ICMP echo request to the target host, and the target host replies with an ICMP echo reply. This can be used to measure latency. We configured Ping to send in total 1000 echo requests with one sent every 10 milliseconds. The other test set consisted of sending TCP traffic with the Iperf tool. The same metrics were used for both test sets:

- There is a huge variation in latency. Depending on connection and how far away a target machine is. This makes it hard to define some standard latencies to test. We wanted to test with some latencies that make trouble for real world applications, as this would be the kinds of delay one would like to emulate when doing tests. There is research being done on online gaming. The different kinds of games have different kinds of delay requirements from the users as shown by Claypool, et. al. in [7]. We chose to use the player tolerance latency for FPS games presented in the paper, of 100 milliseconds. Another set of computer applications that should not exceed a certain amount of latency is Voice over IP (VoIP) applications. An ITU-T recommendation [20] specifies that latency in audio conferences must stay below 150-200 ms to achieve user satisfaction and below 400ms to remain usable. From this we chose to use 400ms as one of the latencies in our test set. In addition, we wanted a relatively short delay to make sure our rate control did not make much difference here. We chose to go with a 20ms RTT.
- We used the same method for calculating queue sizes as in the rate control tests.

This left us with the test scenarios shown in 4.3, which we ran with both Ping and Iperf.

Result

The statistics from the different Ping tests are shown in table 4.4. From the table we can see that the average latency in Double-queue NetEm is very close to NetEm v3.2 in all the tests. This gives an indication that the latency emulation is working as intended. In important thing to notice from the table is that the Dq implementation proposed in this thesis, introduces a higher overhead to the emulation, in the range of 0.12-0.16 milliseconds.

Delay RTT (ms)	Queue size (packets)
20	17
100	84
400	334

Table 4.3: Latency test scenarios

Scenario		Min	Average	Max	Stddev
Emulated delay	Type	(ms)	(ms)	(ms)	(ms)
Reference	none	0.124	0.209	10.258	0.712
20	v3.2	20.115	20.222	30.246	0.840
	Dq	20.267	20.379	30.398	0.899
100	v3.2	100.121	100.236	110.247	0.899
	Dq	100.274	100.361	110.391	0.779
400	v3.2	400.131	400.240	410.241	0.953
	Dq	400.263	400.356	410.387	0.779

Table 4.4: Ping results

The system with two queues and the moving of packets between them is the likely culprit as the calculations done within the queueing discipline has not been changed greatly.

A small problem with the tests can be seen in the Max latency column. All tests have a max latency of 10 milliseconds more than configured. With no other traffic than the ICMP packets themselves, this should not be possible. As we can see from the reference test which we did with no emulation turned on, the problem is still there. This means that it is not a problem induced by the emulation. We narrowed the problem down to the Ping implementation. At random times it seems as if the target machine of the ping request does not send the reply. It will however send it at the same time that it replies to the next ping request. As we sent one request every 10 milliseconds we would get some replies that took that amount of time extra as it had to wait for the next request. We verified this by sending requests at a different rate.

Figure 4.6 shows the results from the TCP test with Iperf. Here we can see the latencies varying with the different RTT scenarios. From the figure we see a common trend that the Dq NetEm implementation overall has a higher average latency and a higher maximum latency than the v3.2 version. This is caused by the fact that Dq NetEm has two queues. Both scenarios were configured with the same queue lengths. In the v3.2 NetEm scenarios both rate control and delay emulation is done with the

same queue. This means that packets waiting because of emulated delay populate the same queue, in other words these packets count as part of the buffer. In Dq NetEm all the packets that are being held back because of delay emulation is placed in the delay queue. This means that these packets do not count towards the queue occupation of the rate queue. The rate queue can therefore hold more packets. With TCP reacting to packet loss, we get a slightly different behaviour with Dq NetEm when compared with v3.2 NetEm.

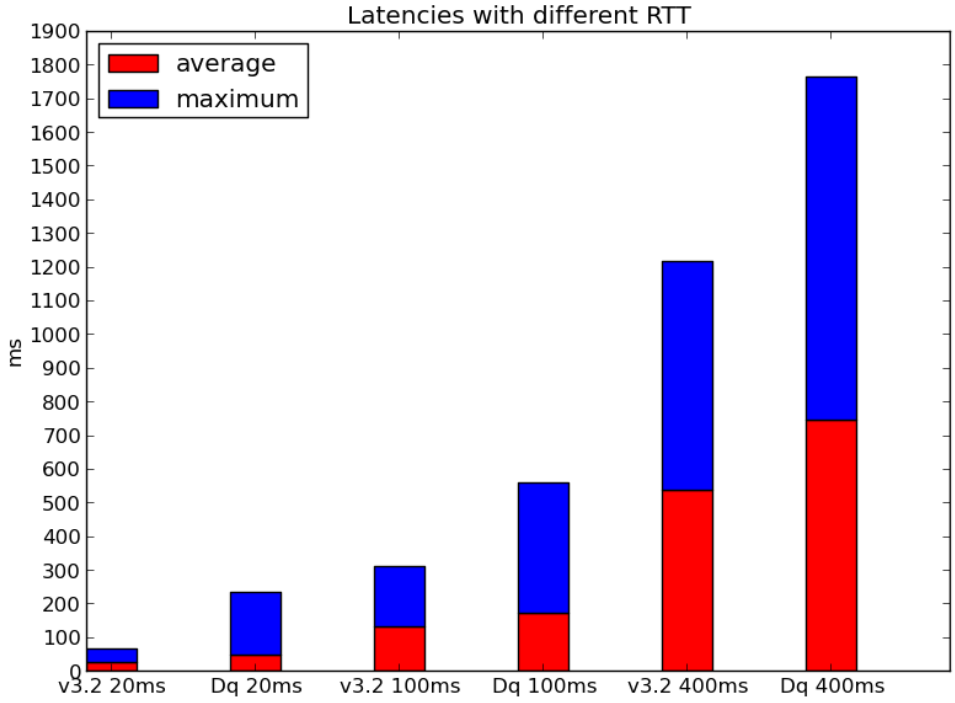


Figure 4.6: Latency vs. RTT

We conclude that the Dq implementation does not have negative impacts on the latency emulation capabilities of NetEm. It does introduce a slight overhead increase, but with the increase being in the range of 0.12-0.16 milliseconds, this is not that big of a problem. We would argument that the difference in behaviour when it comes to the TCP latency tests is a good change. All packets in a TCP stream counts in its packets in flight mechanism, even those out on the physical medium. With one queue we drop packets earlier than we should as our queues also include these packets. This results in earlier drops and a lower packets in flight count than in a real system using the same queue size. In our double queue implementation, the queue size will not include the packets that are staged for delay. This leads to more realistic drops and a higher packets in flight count, which should lead to more realistic behaviour.

4.3 Discussion

As the focus of this thesis has been the implementation of a built-in rate control to NetEm, this is the feature we have extensively tested. We presented some problems with using two queueing disciplines and the resulting single queue in section 3.1. In addition we explained a bit about TC not being developed to be a network emulator. Three main problems were drawn from this: configuration is really hard to get right, using one queue to emulate both rate and delay might not be in the users best interest, and the rate controls based on the token bucket algorithm can easily provide unwanted bursts.

We have seen through the results of the rate control that the rate extension works mostly as expected. We can see a throughput that is very close to what a configured network device gives us. In addition we can plainly see that it leads to less burst than the same set-up used with TBF and NetEm v3.2. Unfortunately it does not seem that we have eliminated burst completely. This might not be possible though, with the inherent burstiness of the underlying architecture. By this we mean that the system is not dedicated to only doing emulation, and that it will at times not manage to run for example the dequeue function at exactly the right time every time it wants to run. When compared to the NetEm v3.8 version we found that they behaved very much alike. We contribute this to the fact that the actual rate emulation is done much in the same way.

Another thing we have seen in the results is the impact of the double-queue system. In the latency experiment we clearly saw a larger difference between NetEm v3.2 and Double-queue NetEm than in any other. We concluded that this was because the packets being held back because of delay emulation does not count towards the queue occupation in our proposed system. This should contribute to the emulator producing more life like results. The same behaviour could probably be reached by calculating and configuring a larger queue size using NetEm v3.2, but by doing that we are back to the other problem of configuration troubles.

Moving from the NetEm v3.2 system that have to use two queueing disciplines that both have to be configured to a system only needing one configuration relieves a lot of problems. When configuring two queueing disciplines we need a great deal of knowledge to get a system that behaves in any way like a real system would. In section 3.1.2 we explained that swapping two queueing disciplines around could completely change the way the system worked. For example, going from a TBF root and NetEm leaf to a NetEm root and TBF leaf changes the reordering mechanisms used in NetEm. This is not something that has a good documentation anywhere, and entails a very good understanding of how the system works. This level of knowledge about how exactly the queueing disciplines work and act when used together is no longer needed when we only have to deal with one qdisc.

Chapter 5

Conclusion and future work

In this thesis we have investigated some of the challenges with using TC and NetEm to do network emulation. This work started out after we experienced some experiments giving weird and unexpected results. After some time we discovered that it was simply the configuration that lead to the strange results. After more time analysing the problem, we found three main problems. 1) Configuration of the queueing disciplines, 2) The problem of cooperation between TC and NetEm. And 3) The fact that the TC system was not developed to be a network emulator.

As a remedy to these problems we proposed a design that would remove the need to configure two queueing disciplines and at the same time divide the emulation into two parts that would remove the problems with doing both rate emulation and delay emulation in the same queue. This solution should mitigate or remove the problems we had found.

We then implemented this system and tested it. The result from these tests show us that it is possible to remove or at least mitigate these problems. One queueing discipline is now doing all the work, which means there is only one to configure. In addition we have shown that the results vary from a system using one queue to do both rate and delay emulation, and our system which uses two separate queues. We have provided an argument for why we think this is a more life like behaviour.

While we have extensively tested that our implementation works fine with both rate and delay emulation, we have not, due to time limitation, tested if the other emulation functions of NetEm works as they should. As most of the actual emulation code from NetEm v3.2 is still the same, we expect them to work as they did in that version. To be certain we would have to do more tests.

We want to conclude that doing work in the Linux kernel is hard. We started out with very little actual knowledge about the kernel, and very much time was spent trying to learn all the networking code needed for this thesis. One real problem is the bad and often very outdated documentation. Despite this we feel that we have implemented something that could be a real contribution if it underwent more tests and tweaks.

5.1 Future work

To submit the code as a patch to the Linux kernel it would have to go through some more extensive testing. Some tweaks might also have to be done concerning corner cases, such as using Double-queue NetEm with rate configured but no delay. There is no reason that the packet should have to be moved between the queues for no reason.

There are lots of possibilities available with the system now using two queues. We could change the implementation to a module based approach, that would enable user to implement their own version of rate control for special needs. The possibility to change the queue used for the rate control would also be a nice addition. It would enable the use of advanced queue schemes, such as RED. However, this would again raise the complexity of the configuration and one should therefore be very careful with how to implement it.

Bibliography

[1] Grenville Armitage and Lawrence Stewart. Some thoughts on emulating jitter for user experience trials. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 157–160. ACM, 2004.

[2] Jon CR Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *Networking, IEEE/ACM Transactions on*, 5(5):675–689, 1997.

[3] Blake, S. et al. RFC 2475: An Architecture for Differentiated Services. http://www.rfc-editor.org/rfc/rfc2475.txt, December 1998.

[4] Braden, et al. RFC 2309: Recommendations on Queue Management and Congestion Avoidance in the Internet. http://www.ietf.org/rfc/rfc2309.txt, April 1998.

[5] Marta Carbone and Luigi Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2):12–20, 2010.

[6] Fabio Checconi, Luigi Rizzo, and Paolo Valente. Qfq: Efficient packet scheduling with tight guarantees. 2012.

[7] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, 2006.

[8] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *Networking, IEEE/ACM Transactions on*, 1(4):397–413, 1993.

[9] The Linux Foundation. iproute2. http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2, nov 2009. [Accessed 15.07-2013].

[10] The Linux Foundation. kernel_flow. http://www.linuxfoundation.org/collaborate/workgroups/networking/kernel_flow, nov 2009. [Accessed 15.07-2013].

[11] The Linux Foundation. Netem. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem, nov 2009. [Accessed 15.07-2013].

[12] The Linux Foundation. sk_buff. http://www.linuxfoundation.org/collaborate/workgroups/networking/sk_buff, nov 2009. [Accessed 17.07-2013].

[13] FreeBSD Handbook. Ipfw. http://www.freebsd.org/doc/handbook/firewalls-ipfw.html, July 2013. [Accessed 14.07-2013].

[14] Thomas Gleixner and Douglas Niehaus. Hrtimers and Beyond: Transforming the Linux Time Subsystems. In *Proceedings of the Linux Symposium, Ottawa, Canada*, volume 1, pages 333–346, 2006.

[15] gnu.org. inetutils. http://www.gnu.org/software/inetutils/. [Accessed 21.07-2013].

[16] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. *ACM SIGCOMM Computer Communication Review*, 29(4):147–160, 1999.

[17] Stephen Hemminger. Network emulation with NetEm. In Martin Pool, editor, *LCA 2005, Australia's 6th national Linux conference (linux.conf.au)*, Sydney NSW, Australia, April 2005. Linux Australia, Linux Australia.

[18] Hubert, B. et al. Linux advanced routing & traffic control howto. http://lartc.org/howto/, 2002. [Accessed 26.06-2013].

[19] International Telecommunication Union (ITU-T). Splitterless asymmetric digital subscriber line (ADSL) transceivers, ITU-T Recommendation G.992.2. 1999.

[20] International Telecommunication Union (ITU-T). One-way Transmission Time, ITU-T Recommendation G.114. 2003.

[21] Audrius Jurgelionis, J-P Laulajainen, Matti Hirvonen, and Alf Inge Wang. An empirical study of netem network emulation functionalities. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pages 1–6. IEEE, 2011.

[22] linux foundation.org. Netem Packet loss + correlation. https://lists.linux-foundation.org/pipermail/netem/2007-September/001156.html, Sep 2007.

[23] Linux IMQ. Intermediate queueing device. http://www.linuximq.net/index.html, January 2013. [Accessed 04.07-2013].

[24] LWN.net. TCP small queues. https://lwn.net/Articles/507065/, jul 2012. [Accessed 20.07-2013].

[25] Martin Devera. Hierachical token bucket. http://luxik.cdi.cz/~devik/qos/htb/, July 2003. [Accessed 26.06-2013].

[26] Netfilter.org. The iptables project. http://www.netfilter.org/projects/iptables/index.html, jul 2013.

[27] NLANR/DAST. Iperf. http://iperf.sourceforge.net/. [Accessed 20.07-2013].

[28] Lucas Nussbaum and Olivier Richard. A comparative study of network link emulators. In *Proceedings of the 2009 Spring Simulation Multiconference*, page 85. Society for Computer Simulation International, 2009.

[29] J. Postel. RFC 791: Internet Protocol. http://tools.ietf.org/html/rfc791, September 1981.

[30] J. Postel. RFC 793: Transmission Control Protocol. http://www.ietf.org/rfc/rfc793.txt, September 1981.

[31] Ramakrishnan, et al. RFC 3168: The Addition of Explicit Congestion Notification (ECN) to IP. http://www.ietf.org/rfc/rfc3168.txt, September 2001.

[32] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.

[33] S Salsano, F Ludovici, and A Ordine. Definition of a general and intuitive loss model for packet networks and its implementation in the netem module in the linux kernel. Technical report, Technical report, University of Rome? Tor Vergata, 2009.

[34] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. *ACM SIGCOMM Computer Communication Review*, 25(4):231–242, 1995.

[35] The Linux Kernel Archives. FIFO. https://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/tree/net/sched/sch_fifo.c?id=v3.8, feb 2013. [Accessed 29.06-2013].

[36] The Linux Kernel Archives. HTB. https://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/tree/net/sched/sch_htb.c?id=v3.8, feb 2013. [Accessed 02.07-2013].

[37] The Linux Kernel Archives. Ingress. https://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/tree/net/sched/sch_ingress.c?id=v3.8, feb 2013. [Accessed 03.07-2013].

[38] The Linux Kernel Archives. NetEm. https://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/tree/net/sched/sch_netem.c?id=v3.8, feb 2013. [Accessed 03.07-2013].

[39] The Linux Kernel Archives. pfifo_fast. https://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/tree/net/sched/sch_generic.c?id=v3.8, feb 2013. [Accessed 29.06-2013].

[40] The Linux Kernel Archives. PRIO. https://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/tree/net/sched/sch_prio.c?id=v3.8, feb 2013. [Accessed 01.07-2013].

[41] The Linux Kernel Archives. RED. https://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/tree/net/sched/sch_red.c?id=v3.8, feb 2013. [Accessed 30.06-2013].

[42] The Linux Kernel Archives. TBF. https://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/tree/net/sched/sch_tbf.c?id=v3.8, feb 2013. [Accessed 01.07-2013].

[43] Klaus Wehrle, Mesut Günes, and James Gross, editors. *Modeling and Tools for Network Simulation*. Springer, 2010.